Si no usamos headings no tendremos TOC !!! **(TOC = tabla de contenido o índice)**
Los nombres de archivo y métodos en *itálica*.
El código en una tabla. (No hemos encontrado mejor solución), por ahora se colorea con http://codepad.org y fuente `Courier New`.
El texto y headings en Arial.

Manual de KumbiaPHP Framework Versión 1.0 Spirit

Borrador para la beta2 Final

12 de junio de 2013

www.kumbiaphp.com

Índice general

(no tocar, se crea automático con los headings)

1 Introducción

1.1. Agradecimientos

Este manual es para agradecer a los que con su tiempo y apoyo, en gran o en poca medida, han ayudado a que este framework sea cada día mejor. A toda la comunidad que rodea a KumbiaPHP, con sus preguntas, notificaciones de errores (Bug's), aportes, críticas, etc., a todos ellos ¡Gracias!.

1.2. Prefacio

1.2.1. Sobre este libro

El libro de KumbiaPHP intenta comunicar, todo lo que este framework puede ayudar en su trabajo diario como desarrollador. Le permite descubrir todos los aspectos de KumbiaPHP y aprender porque KumbiaPHP puede ser la herramienta, que estaba esperando para empezar a desarrollar su proyecto. Este libro se encuentra en etapa de continuo desarrollo, diseño gráfico, revisión ortográfica y gramática, contenidos, etc. Tal como sucede con el framework, por lo cual se aconseja siempre disponer de la última versión.

Esta versión del manual ha cambiado mucho de la anterior. Gracias a la comunidad se han reflejado cuestiones que se repetían en grupo, foro e IRC. También se detectó el mal uso del MVC, y no se aprovechaban las facilidades del POO. Se ha intentado mejorar esos puntos recurrentes de consulta, así como mejorar el entendimiento de uso, para que creen mejores y más mantenibles aplicaciones.

1.2.2. Sobre KumbiaPHP

KumbiaPHP es un producto latino para el mundo. Programar debe ser tan bueno como bailar y KumbiaPHP es un baile, un baile para programar. KumbiaPHP es un framework de libre uso bajo **licencia new BSD**. Por lo tanto, puedes usar KumbiaPHP para tus proyectos siempre y cuando tengas en cuenta la licencia. Te aconsejamos que siempre uses versiones estables y lo más recientes posibles, ya que se incluyen correcciones, nuevas funcionalidades y otras mejoras interesantes.

1.2.3. Sobre la comunidad

La comunidad de KumbiaPHP está formada en su gran mayoría por gente hispano-latina, de la cual nace un framework completamente en español. Y donde radica su mayor diferencia respecto a otros frameworks que son, de forma nativa, anglosajones. Por otra parte se ha intentado, con el tiempo, aportar nuevos sistemas de comunicación, así que se cuenta con una lista de correo, el foro, canal de IRC y una WiKi. Esperamos que todo esto haga que la comunidad sea una parte importante y vital para enriquecer y mejorar KumbiaPHP.
Puedes encontrar más información en www.kumbiaphp.com

1.2.4. ¿Porque usar KumbiaPHP Framework?

Mucha gente pregunta ¿cómo es este framework?, ¿otro más?, ¿será fácil? ¿qué tan potente es? etc. Pues aquí algunas razones para utilizar KumbiaPHP:

1. Es muy *fácil de usar* (Zero-Config). Empezar con KumbiaPHP es demasiado fácil, es solo descomprimir y empezar a trabajar, esta filosofía también es conocida como Convención sobre Configuración.
2. *Agiliza el Trabajo*, crear una aplicación muy funcional con KumbiaPHP es cuestión de horas o minutos, así que podemos darle gusto a nuestros clientes sin que afecte nuestro tiempo. Gracias a las múltiples herramientas que proporciona el framework para agilizar el trabajo podemos hacer más en menos tiempo.
3. *Separar la Lógica de la Presentación*, una de las mejores prácticas de desarrollo orientado a la Web es separar la lógica de los datos y la presentación, con KumbiaPHP es sencillo aplicar el patrón MVC (Modelo, Vista, Controlador) y hacer nuestras aplicaciones más fáciles de mantener y de escalar.
4. *Reducción del uso de otros Lenguajes*, gracias a los *Helpers* y a otros patrones como ActiveRecord evitamos el uso de lenguajes HTML y SQL (en menor porcentaje). KumbiaPHP hace esto por nosotros, con esto logramos código mas claro, natural y con menos errores.
5. *¡Habla Español!* La documentación, mensajes de error, archivos de configuración, comunidad, desarrolladores ¡hablan español!, con esto no tenemos que entender a medias, como con otros frameworks que nos toca quedarnos cruzados de manos porque no podemos pedir ayuda.
6. *La Curva de Aprendizaje* de KumbiaPHP es muy corta, y si a esto le agregamos experiencia en el manejo de Programación Orientada a Objetos, será mas rápida.
7. *Parece un juego*, sin darnos cuenta aplicamos los patrones de diseño; los patrones de diseño son herramientas que facilitan el trabajo realizando abstracción, reduciendo código o haciendo más fácil de entender la aplicación. Cuando trabajas con KumbiaPHP aplicas muchos patrones y al final te das cuenta que eran más fáciles de usar de lo que se piensa.
8. *Software Libre*, No tienes que preocuparte por licenciar nada, KumbiaPHP promueve las comunidades de aprendizaje, el conocimiento es de todos y cada uno sabe como aprovecharlo mejor.
9. *Aplicaciones Robustas*, ¿no sabía que tenía una?. Las aplicaciones de hoy día requieren arquitecturas robustas. KumbiaPHP proporciona una arquitectura fácil de aprender y de implementar, sin complicarnos con conceptos y sin sacrificar calidad.

1.2.5. ¿Qué aporta KumbiaPHP?

KumbiaPHP es un esfuerzo por producir un framework que ayude a reducir el tiempo de desarrollo de una aplicación web sin producir efectos sobre los programadores, basándonos en principios claves, que siempre recordamos.

- *KISS* «Mantenlo simple, estúpido» (Keep It Simple, Stupid).
- *DRY* No te repitas, en inglés Don’t Repeat Yourself, también conocido como Una vez y

sólo una.
- Convención sobre configuración.
- Velocidad.

Además KumbiaPHP está fundamentado en las siguientes premisas:

- Fácil de aprender.
- Fácil de instalar y configurar.
- Compatible con muchas plataformas.
- Listo para aplicaciones comerciales.
- Simple en la mayor parte de casos pero flexible para adaptarse a casos más complejos.
- Soportar muchas características de aplicaciones Web actuales.
- Soportar las prácticas y patrones de programación más productivos y eficientes.
- Producir aplicaciones fáciles de mantener.
- Basado en Software Libre.

Lo principal es producir aplicaciones que sean prácticas para el usuario final y no sólo para el programador. La mayor parte de tareas que le quiten tiempo al desarrollador deberían ser automatizadas por KumbiaPHP, para que pueda enfocarse en la lógica de negocio de su aplicación. No deberíamos reinventar la rueda cada vez que se afronte un nuevo proyecto de software.

Para satisfacer estos objetivos KumbiaPHP está escrito en PHP5. Además ha sido probado en aplicaciones reales que trabajan en diversas áreas con variedad de demanda y funcionalidad. Es compatible con las bases de datos disponibles actuales mas usadas:

- MySQL.
- PostgreSQL.
- Oracle.
- SQLite.

El modelo de objetos de KumbiaPHP es utilizado en las siguientes capas:

- Abstracción de la base de datos.
- Mapeo Objeto-Relacional.
- Modelo MVC (Modelo, Vista, Controlador).

Características comunes de aplicaciones Web que son automatizadas por KumbiaPHP:

- Plantillas (TemplateView).
- Validación y Persistencia de Formularios.
- Administración de Caché.
- Scaffolding.
- Front Controller.
- Interacción AJAX.
- Generación de Formularios.

- Seguridad.

1.2.6. Modelo, Vista, Controlador (MVC)

1.2.6.1. ¿Que es MVC?

En 1979, Trygve Reenskaug desarrolló una arquitectura para desarrollar aplicaciones interactivas. En este diseño existían tres partes: modelos, vistas y controladores. El modelo MVC permite hacer la separación de las capas de interfaz, modelo y lógica de control de ésta. La programación por capas, es un estilo de programación en la que el objetivo primordial es la separación de la lógica de negocios de la lógica de diseño, un ejemplo básico de esto es separar la capa de datos de la capa de presentación al usuario.

La ventaja principal de este estilo, es que el desarrollo se puede llevar a cabo en varios niveles y en caso de algún cambio sólo se ataca al nivel requerido sin tener que revisar entre código mezclado. Además permite distribuir el trabajo de creación de una aplicación por niveles, de este modo, cada grupo de trabajo está totalmente abstraído del resto de niveles, simplemente es necesario conocer la API (Interfaz de Aplicación) que existe entre niveles. La división en capas reduce la complejidad, facilita la reutilización y acelera el proceso de ensamblar o desensamblar alguna capa, o sustituirla por otra distinta (pero con la misma responsabilidad).

En una aplicación Web una petición se realiza usando HTTP y es enviado al controlador. El controlador puede interactuar de muchas formas con el modelo, luego él primero llama a la respectiva vista la cual obtiene el estado del modelo que es enviado desde el controlador y lo muestra al usuario.

1.2.6.2. ¿Como KumbiaPHP aplica el MVC?

KumbiaPHP Framework aprovecha los mejores patrones de programación orientada a la Web en especial el patrón MVC (Modelos, Vistas, Controladores). A continuación se describe el funcionamiento general de este paradigma en KumbiaPHP.

El objetivo de este patrón es el realizar y mantener la separación entre la lógica de nuestra aplicación, los datos y la presentación. Esta separación tiene algunas ventajas importantes, como poder identificar más fácilmente en qué capa se está produciendo un problema con sólo conocer su naturaleza. Podemos crear varias presentaciones sin necesidad de escribir varias veces la misma lógica de aplicación. Cada parte funciona independiente y cualquier cambio centraliza el efecto sobre las demás, así que podemos estar seguros que una modificación en un componente realizará bien las tareas en cualquier parte de la aplicación.

1.2.6.3. Más información

La base de KumbiaPHP es el MVC y POO, un tradicional patrón de diseño que funciona en tres capas:

- **Modelos:** Representan la información sobre la cual la aplicación opera, su lógica de negocio.
- **Vistas:** Visualizan el modelo usando páginas Web e interactuando con los usuarios (en principio) de éstas, una vista puede estar representada por cualquier formato salida, nos referimos a un xml, pdf, json, svg, png, etc. todo esto son vistas.

- **Controladores:** Responden a acciones de usuario e invocan cambios en las vistas o en los modelos según sea necesario.

En KumbiaPHP los controladores están separados en partes, llamadas front controller y en un conjunto de acciones. Cada acción sabe cómo reaccionar ante un determinado tipo de petición.

Las vistas están separadas en templates, views y partials.

El modelo ofrece una capa de abstracción de la base de datos, además da funcionalidad agregada a datos de sesión y validación de integridad relacional. Este modelo ayuda a separar el trabajo en lógica de negocios (Modelos) y la presentación (Vistas).

Por ejemplo, si usted tiene una aplicación que corra tanto en equipos de escritorio y en dispositivos móviles entonces podría crear dos vistas diferentes compartiendo las mismas acciones en el controlador y la lógica del modelo. El controlador ayuda a ocultar los detalles de protocolo utilizados en la petición (HTTP, modo consola, etc.) para el modelo y la vista.

Finalmente, el modelo abstrae la lógica de datos, que hace a los modelos independientes de las vistas. La implementación de este modelo es muy liviana mediante pequeñas convenciones se puede lograr mucho poder y funcionalidad.

1.2.6.4. Caso práctico (CAMBIAR EJEMPLO)

Para entender mejor, veamos un ejemplo de cómo una arquitectura MVC trabaja para añadir al carrito. Primero, el usuario interactúa con la interfaz seleccionando un producto y presionando un botón, esto probablemente valida un formulario y envía una petición al servidor.

1. El Front Controller recibe la notificación de una acción de usuario, y luego de ejecutar algunas tareas (enrutamiento, seguridad, etc.), entiende que debe ejecutar la acción de agregar en el controlador.
2. La acción de agregar accede al modelo y actualiza el objeto del carrito en la sesión de usuario.
3. Si la modificación es almacenada correctamente, la acción prepara el contenido que será devuelto en la respuesta – confirmación de la adición y una lista completa de los productos que están actualmente en el carrito. La vista ensambla la respuesta de la acción en el cuerpo de la aplicación para producir la página del carrito de compras.
4. Finalmente es transferida al servidor Web que la envía al usuario, quien puede leerla e interactuará con ella de nuevo.

2 Empezando con KumbiaPHP

2.1. Instalar KumbiaPHP

En esta sección, se explican los pasos a seguir, para poner a funcionar el framework en nuestro ambiente de desarrollo.

2.1.1. Requisitos

Como se mencionó arriba KumbiaPHP es muy fácil y en este sentido los requerimientos para hacer funcionar el framework son mínimos, a continuación se listan:

- Intérprete PHP (versión 5.2.2 o superior).
- Servidor Web con soporte de reescritura de URL (Apache, Cherokee, Lighttpd, Internet Information Server (IIS)).
- Manejador de base de datos soportado por KumbiaPHP.

Para instalar KumbiaPHP Framework, se debe descargar su archivo comprimido desde la sección de descarga http://www.kumbiaphp.com/blog/manuales-y-descargas/ para obtener la versión más reciente del framework. Una vez descargado el archivo, es esencial asegurarse que tiene la extensión .tgz para usuarios Linux y .zip para usuarios de Windows, ya que de otro modo no se descomprimirá correctamente.

A continuación se descomprime su contenido en el directorio raíz del servidor web (DocumentRoot). Para asegurar cierta uniformidad en el documento, en este capítulo se supone que se ha descomprimido el paquete del framework en el directorio *kumbiaphp/*. Teniendo una estructura como la siguiente:

```
`-- 1.0
   |-- core
   |-- default
      |-- app
      |-- index.php
      |-- .htaccess
      `-- public
```

2.1.2. Configurar Servidor Web

KumbiaPHP Framework utiliza un módulo para la reescritura de URLs haciéndolas más comprensibles y fáciles de recordar en nuestras aplicaciones. Por esto, el módulo debe ser configurado e instalado, en este sentido debe chequear que el módulo esté habilitado, en las siguientes secciones se explica como hacer.

2.1.2.1. Habitando mod_rewrite de Apache en GNU/Linux (Debian, Ubuntu y Derivados)

Nos aseguramos de activar el *mod_rewrite* de esta manera y como usuario administrador desde la consola.

```
#a2enmod rewrite
Enabling module rewrite.
Run '/etc/init.d/apache2 restart' to activate new configuration!
```

Lo anterior indica que se ha habilitado el *mod_rewrite* de Apache, pero aun falta indicarle a Apache que interprete los archivos *.htaccess* que son los encargados de hacer uso del rewrite y a su vez tienen las reglas de reescritura de las URLs.

Como usuario administrador editamos el siguiente archivo.

```
#vi /etc/apache2/sites-enabled/000-default
```

```
<Directory "/to/document/root">
   Options Indexes FollowSymLinks
   AllowOverride None
   Order allow,deny
   Allow from all
</Directory>
```

Para que los .htaccess tengan efectos, se ha de sustituir AllowOverride None por AllowOverride All, de esta manera Apache podrá interpretar estos archivos.
Hecho esto, queda reiniciar el servicio de apache.

```
#/etc/init.d/apache2 restart
```

A continuación, se prueba todas las configuraciones realizadas mediante la siguiente URL.

```
http://localhost/kumbiaphp/
```

Si todo ha ido bien, deberías ver una página de bienvenida como la que se muestra en la figura 2.1, con lo que la instalación rápida se puede dar por concluida.



Figura 2.1: Instalación Exitosa de KumbiaPHP

Esto es un entorno de pruebas el cual está pensado para que practiques con KumbiaPHP en un servidor local, no para desarrollar aplicaciones complejas que terminan siendo publicadas en la web.

2.1.2.2. ¿Por qué es importante el Mod-Rewrite?

ReWrite es un módulo de apache que permite reescribir las urls que han utilizado nuestros usuarios. KumbiaPHP Framework encapsula esta complejidad permitiéndonos usar URLs bonitas o limpias como las que vemos en blogs o en muchos sitios donde no aparecen los ?, los & ni las extensiones del servidor (.php, .asp, .aspx, etc).

Además de esto, con *mod-rewrite* KumbiaPHP puede proteger nuestras aplicaciones ante la posibilidad de que los usuarios puedan ver los directorios del proyecto y puedan acceder a archivos de clases, modelos, lógica, etc., sin que sean autorizados.

Con *mod-rewrite* el único directorio que pueden ver los usuarios es el contenido del directorio público (public) del servidor web, el resto permanece oculto y sólo puede ser visualizado cuando ha realizado una petición en forma correcta y también es correcto según nuestra lógica de aplicación. Cuando escribes direcciones utilizando este tipo de URLs, estás ayudando también a los motores de búsqueda a indexar mejor tu información.

2.1.3. Mi Primera Aplicación con KumbiaPHP

Luego que explicamos los pasos para configurar KumbiaPHP y ver su pantalla de bienvenida, se viene hacer el primer ejemplo el cual tiene como objetivo entender elementos básicos al momento de utilizar el framework que servirá para entender la arquitectura MVC (Modelo-Vista-Controlador).

2.1.3.1. Hola, KumbiaPHP!

Ahora escribiremos el famoso "Hola, Mundo!" pero con un pequeño cambio: Diremos "Hola, KumbiaPHP!". Pensando en esto, recordemos el modelo MVC, según esto, KumbiaPHP debería aceptar una petición, que buscaría en controlador y, en éste, una acción que atendería la petición. Luego, KumbiaPHP utilizará esta información de controlador y acción para buscar la vista asociada a la petición.

Para escribir el código de nuestro "Hola, KumbiaPHP!" no necesitamos sino un controlador y una vista. No necesitamos modelos, ya que no estamos trabajando con información de una base de datos.

Nos ubicamos en el directorio /path/to/kumbiaphp/app/controllers/. Aquí estarán nuestros controladores (Para más detalles, lee la documentación sobre el directorio app). Para crear un controlador, es importante tener en cuenta las convenciones de nombre que utiliza el Framework. llamaremos a nuestro controlador *saludo_controller.php*. Nótese el sufijo _controller.php esto forma parte de la convención de nombres, y hace que KumbiaPHP identifique ese archivo como un controlador.

2.1.3.2. El Controlador

Ahora agregamos contenido al controlador *app/controllers/saludo_controller.php*

```
<?php
/**
 * Controller Saludo
 */
class SaludoController extends AppController {
    public function hola() {
    }
}
```

En el código tenemos la definición de la class *SaludoController*, Nótese que también está el sufijo Controller al final de la declaración de la clase, esto la identifica como una clase controladora, y ésta hereda (extends) de la superclase *AppController*, con lo que adquiere las propiedades de una clase controladora, además existe el método *hola()*.

2.1.3.3. La Vista

Para poder ver la salida que envía el controlador, es necesario crear la vista asociada a la acción. Primero, creamos un directorio con el mismo nombre de nuestro controlador (en este caso debería llamarse saludo), y dentro de este estarán todas las vistas asociadas a las acciones que necesiten mostrar alguna información. En nuestro ejemplo llamamos a una acción llamada hola; por lo tanto, creamos un archivo llamado *app/views/saludo/hola.phtml*. Una vez creado este archivo, le agregamos un poco de contenido:

```
<h1>Hola, KumbiaPHP!</h1>
```

A continuación se prueba al acceder a la siguiente URL: http://localhost/ kumbiaphp/saludo/hola/ y el resultado debe ser como muestra la figura 2.2.



Figura 2.2: Contenido de la vista hola.phtml

2.1.3.4. KumbiaPHP y sus URLs

Para entender el funcionamiento del framework es importante entender sus URLs, la figura 2.3muestra una URL típica en KumbiaPHP.



Figura 2.3: URL en KumbiaPHP

En KumbiaPHP no existen las extensiones .php esto porque en primera instancia hay reescritura de URLs y además cuenta con un front-controller encargado de recibir todas las peticiones (más adelante se explicará en detalle).

Cualquier otra información pasada por URL es tomada como parámetro de la acción, a propósito de nuestra aplicación como muestra la figura 2.4.

http://localhost/kumbiaphp/saludo/hola/param1/param2/

1) Parámetro 1
2) Parámetro 2

1 2

```
...
    public function hola($param1, $param2)
    {
      //Utilizan los parámetros enviados
    }
...
```

Figura 2.4: URL con parámetros

Esto es útil para evitar que tener estar enviando parámetros GET de la forma ?var=valor&var2=valor2 (esto es, de la forma tradicional como se viene utilizando PHP), la cual revela información sobre la arquitectura de software que se dispone en el servidor. Además, hacen que nuestra URL se vea mal.

2.1.3.5. Agregando más contenido

Agregaremos algo de contenido dinámico a nuestro proyecto, para que no sea tan aburrido. Mostraremos la hora y la fecha, usando la función *date()*.
Cambiamos un poco el controlador *app/controllers/saludo_controller.php*

```php
<?php
/**
 * Controller Saludo
 */
class SaludoController extends AppController {
    /**
     * método para saludar
     */
    public function hola() {
        $this->fecha = date("Y-m-d H:i");
    }
}
```

KumbiaPHP implementa las variables de instancia lo que significa que todos atributos definidos en el controller, pasará automáticamente a la vista, en el código anterior tenemos el atributo *$this->fecha* este pasará a la vista como una variable llamada *$fecha*.

En la vista que se creo en la sección 2.1.3.3 y agregamos.

```
<h1>Hola, KumbiaPHP!</h1>
<?php echo $fecha ?>
```

Ahora, si volvemos a http://localhost/kumbiaphp/saludo/hola/, obtendremos la hora y fecha del momento en que se haga la petición, como se muestra en la figura 2.5.



Figura 2.5: Hora y fecha de petición

Para agregarle calidez al asunto, le preguntaremos al usuario su nombre vía parámetro 2.1.3.4, volvemos a editar el controlador *saludo_controller.php...*

```
<?php
/**
 * Controller Saludo
 */
class SaludoController extends AppController
{
    /**
     * método para saludar
     * @param string $nombre
     */
    public function hola($nombre)
    {
        $this->fecha = date("Y-m-d H:i");
        $this->nombre = $nombre;
    }
}
```

Editamos la vista *app/views/saludo/hola.phtml*

```
<h1>Hola <?php echo $nombre; ?>, ¡Qué lindo es utilizar KumbiaPHP!
¿cierto?</h1>
<?php echo $fecha ?>
```

Si ahora entramos a http://localhost/kumbiaphp/saludo/CaChi/, nos mostrará en el navegador web el saludo junto con el nombre colocado y la fecha actual, como se muestra en la figura 2.6.

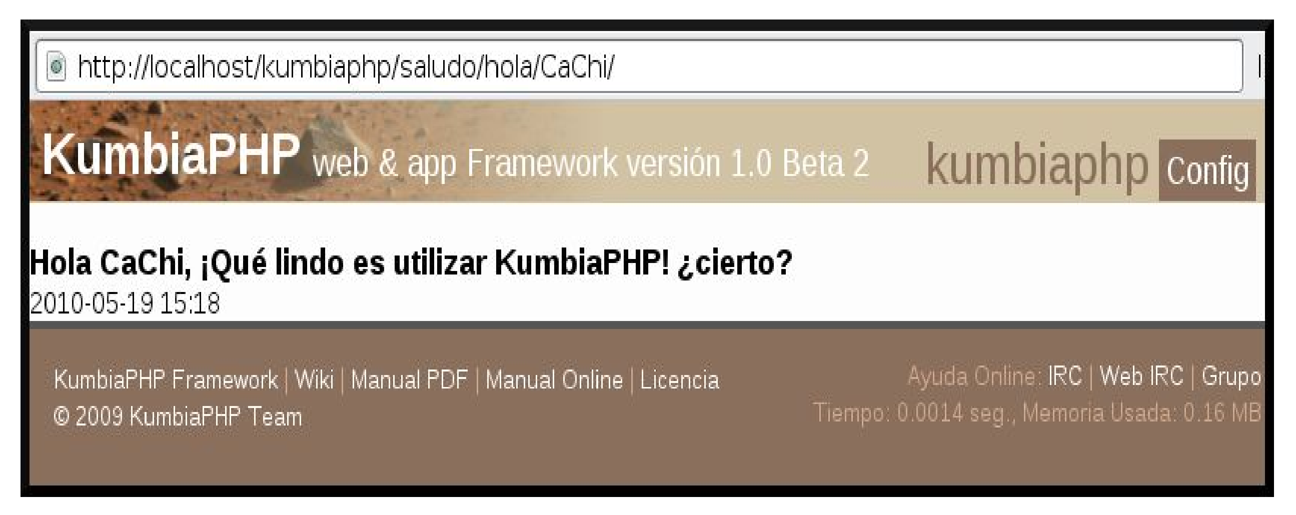


Figura 2.6: Saludando al Usuario

2.1.3.6. Repitiendo la Historia

Ahora vamos otra acción llamada *adios()* y como su nombre indica haremos el proceso inverso a saludar, es decir despedir a los usuarios.

```
<?php
/**
 * Controller Saludo
 */
class SaludoController extends AppController {
    /**
     * método para saludar
     * @param string $nombre
     */
    public function hola($nombre) {
        $this->fecha = date("Y-m-d H:i");
        $this->nombre = $nombre;
    }
    /**
     * método para despedir
     */
    public function adios() {
    }
}
```

Agregamos una nueva vista para presentar el contenido de la acción *adios()* y si recordamos lo que se explicó en la sección 2.1.3.3 deberíamos crear una vista *app/views/saludo/adios.phtml* con el siguiente contenido.

```
<h1>Ops! se ha ido :( </h1>
<?php echo Html::link('saludo/hola/CaChi/' , 'Volver a Saludar' ); ?>
```

Si ingresa al siguiente enlace http://localhost/kumbiaphp/saludo/adios/ se verá un nuevo texto, y un vínculo a la acción *hola()*, como se muestra en la figura 2.7.



Figura 2.7: Vista de adiós al usuario.

Html::link(), es uno de los tantos helper que ofrece KumbiaPHP para facilitar al momento de programar en las vistas. Podríamos escribir el código HTML directamente, colocando *Volver a Saludar*, pero esto puede conllevar a un problema, imagine que quisiera cambiar de nombre a su proyecto de kumbiaphp a demo, tendríamos que modificar todos los vínculos, los helpers de KumbiaPHP resuelven estos problemas.

3 El Controlador

En KumbiaPHP Framework, la capa del controlador, contiene el código que liga la lógica de negocio con la presentación, está dividida en varios componentes que se utilizan para diversos propósitos:

- El controlador frontal (front controller) es el único punto de entrada a la aplicación. Carga la configuración y determina la acción a ejecutarse.
- Las acciones verifican la integridad de las peticiones y preparan los datos requeridos por la capa de presentación.
- Las clases Input y Session dan acceso a los parámetros de la petición y a los datos persistentes del usuario. Se utilizan muy a menudo en la capa del controlador.
- Los filtros son trozos de código ejecutados para cada petición, antes y/o después de un controlador incluso antes y/o después de una acción. Por ejemplo, los filtros de seguridad y validación son comúnmente utilizados en aplicaciones web.

,

Este capítulo describe todos estos componentes. Para una página básica, es probable que solo necesites escribir algunas líneas de código en la clase de la acción, y eso es todo. Los otros componentes del controlador solamente se utilizan en situaciones específicas.

3.1. Controlador Frontal

Todas las peticiones web son manejadas por un solo Controlador Frontal (front controller), que es el punto de entrada único de toda la aplicación.

Cuando el front controller recibe la petición, utiliza el sistema de enrutamiento de KumbiaPHP para asociar el nombre de un controlador y el de la acción mediante la URL escrita por el cliente (usuario u otra aplicación).

Veamos la siguiente URL, ésta llama al script *index.php* (que es el front controller) y será entendido como llamada a una acción.

```
http://localhost/kumbiaphp/micontroller/miaccion/
```

Debido a la reescritura de URL nunca se hace un llamado de forma explícita al *index.php*, sólo se coloca el controlador, acción y parámetros. Internamente por las reglas reescritura de URL es llamado el front controller. Ver sección ¿por qué es importante el Mod-Rewrite?

3.1.1. Destripando el Front Controller

El front controller de KumbiaPHP se encarga de despachar las peticiones, lo que implica algo más que detectar la acción que se ejecuta. De hecho, ejecuta el código común a todas las acciones, incluyendo:

1. Define las constantes del núcleo de la aplicación(APP_PATH,CORE_PATH y

PUBLIC_PATH).
2. Carga e inicializa las clases del núcleo del framework (bootstrap).
3. Carga la configuración (Config).
4. Decodifica la URL de la petición para determinar la acción a ejecutar y los parámetros de la petición (Router).
5. Si la acción no existe, redireccionará a la acción del error 404 (Router).
6. Activa los filtros (por ejemplo, si la petición necesita autenticación) (Router).
7. Ejecuta los filtros, primera pasada (before). (Router)
8. Ejecuta la acción (Router).
9. Ejecuta los filtros, segunda pasada (after) (Router).
10. Ejecuta la vista y muestra la respuesta (View).

En grande rasgos éste es el proceso del front controller, esto es todo que necesitas saber sobre este componente el cual es imprescindible de la arquitectura MVC dentro de KumbiaPHP

3.1.2. Front Controller por defecto

El front controller por defecto, llamado *index.php* y ubicado en el directorio *public/* del proyecto, es un simple script, como el siguiente:

```
...
error_reporting(E_ALL ^ E_STRICT);
...
//define('PRODUCTION', TRUE);
...
define('START_TIME', microtime(1));
...
define('APP_PATH', dirname(dirname(__FILE__)) . '/app/');
...
define('CORE_PATH', dirname(dirname(APP_PATH)) . '/core/');
...
if ($_SERVER['QUERY_STRING']) {
    define('PUBLIC_PATH', substr(urldecode($_SERVER['REQUEST_URI']),
0, - strlen($_SERVER['QUERY_STRING']) + 6));
} else {
   define('PUBLIC_PATH', $_SERVER['REQUEST_URI']);
}
...
 $url = isset($_GET['_url']) ? $_GET['_url'] : '/';
...
require CORE_PATH . 'kumbia/bootstrap.php';
```

La definición de las constantes corresponde al primer paso descrito en la sección anterior. Después el controlador frontal incluye el *bootstrap.php* de la aplicación, que se ocupa de los pasos 2 a 5. Internamente el core de KumbiaPHP con sus componente Router y View ejecutan todos los pasos subsiguientes.

Todas las constantes son valores por defecto de la instalación de KumbiaPHP en un ambiente local.

3.1.3. Constantes de KumbiaPHP

Cada constante cumple un objetivo específico con el fin de brindar mayor flexibilidad al momento de crear rutas (paths) en el framework.

3.1.3.1. APP_PATH

Constante que contiene la ruta absoluta al directorio donde se encuentra la aplicación (app), por ejemplo:

```
echo APP_PATH;
//la salida es: /var/www/kumbiaphp/default/app/
```

Con esta constante es posible utilizarla para incluir archivos que se encuentre bajo el árbol de directorio de la aplicación, por ejemplo si quiere incluir un archivo que está en el directorio *app/libs/test.php* la forma de hacerlo sería.

```
include_once APP_PATH.'libs/test.php';
```

3.1.3.2. CORE_PATH

Constante que contiene la ruta absoluta al directorio donde se encuentra el core de KumbiaPHP. por ejemplo:

```
echo CORE_PATH;
//la salida es: /var/www/kumbiaphp/core/
```

Para incluir archivos, que se encuentre bajo este árbol de directorios, es el mismo procedimiento que se explicó para la constante APP_PATH.

3.1.3.3. PUBLIC_PATH

Constante que contiene la URL para el navegador (browser) y apunta al directorio *public/* para enlazar imágenes, CSS, JavaScript y todo lo que sea ruta para browser.

```
//Genera un link que ira al
//controller: controller y action: action
<a href="<?php echo PUBLIC_PATH ?>controller/action/" title="Mi
Link">Mi Link</a>

//Enlaza una imagen que esta en public/img/imagen.jpg
<img src="<?php echo PUBLIC_PATH ?>img/imagen.jpg" alt="Una Imagen"
```

```
/>

//Enlaza el archivo CSS en public/css/style.css
<link rel="stylesheet" type="text/css" href=" <?php echo PUBLIC_PATH
?>css/style.css"/>
```

3.2. Las Acciones

Las acciones son la parte fundamental en la aplicación, puesto que contienen el flujo en que la aplicación actuará ante ciertas peticiones. Las acciones utilizan el modelo y definen variables para la vista. Cuando se realiza una petición web en una aplicación KumbiaPHP, la URL define una acción y los parámetros de la petición. Ver sección 2.1.3.4

Las acciones son métodos de una clase controladora llamada ClassController que hereda de la clase *AppController* y pueden o no ser agrupadas en módulos.

3.2.1. Las acciones y las vistas

Cada vez que se ejecuta una acción KumbiaPHP buscará entonces una vista (view) con el mismo nombre de la acción. Este comportamiento se ha definido por defecto. Normalmente las peticiones deben dar una respuesta al cliente que la ha solicitado, entonces si tenemos una acción llamada *saludo()* debería existir una vista asociada a esta acción llamada *saludo.phtml*. Habrá un capítulo más extenso dedicado a la explicación de las vistas en KumbiaPHP.

3.2.2. Obtener valores desde una acción

Las URLs de KumbiaPHP están caracterizadas por tener varias partes, cada una de ellas con una función conocida. Para obtener desde un controlador los valores que vienen en la URL podemos usar algunas propiedades definidas en el controlador.

Tomemos la URL:

```
http://www.dominio.com/noticias/ver/12/
```

- **El Controlador:** *noticias*
- **La acción:** *ver*
- **Parámetros:** *12*

```
<?php
/**
 * Controller Noticia
 */
class NoticiasController extends AppController{
    /**
     * método para ver la noticia
```

```
     * @param int $id
     */
    public function ver($id){
        echo $this->controller_name; //noticias
        echo $this->action_name; //ver
        //Un array con todos los parámetros enviados a la acción
        var_dump($this->parameters);
    }
}
```

Es importante notar la relación que guardan los parámetros enviados por URL con la acción. En este sentido KumbiaPHP tiene una característica, que hace seguro el proceso de ejecutar las acciones y es que se limita el envío de parámetros tal como se define en la método (acción). Lo que indica que todos los parámetros enviados por URL son argumentos que recibe la acción. ver sección 2.1.3.4

En el ejemplo anterior se definió en la acción ver($id) un solo parámetro, esto quiere decir que si no se envía ese parámetro o se intentan enviar más parámetros adicionales KumbiaPHP lanzará una exception (en producción mostrará un error 404). Este comportamiento es por defecto en el framework y se puede cambiar para determinados escenarios según el propósito de nuestra aplicación para la ejecución de una acción.

Tomando el ejemplo «Hola Mundo» ponga en práctica lo antes explicado y lo hará enviando parámetros adicionales al método hola($nombre) el cual sólo recibe un sólo parámetro (el nombre) http://localhost/kumbiaphp/saludo/hola/CaChi/adicional, en la figura 3.1 verá la excepción generada por KumbiaPHP.

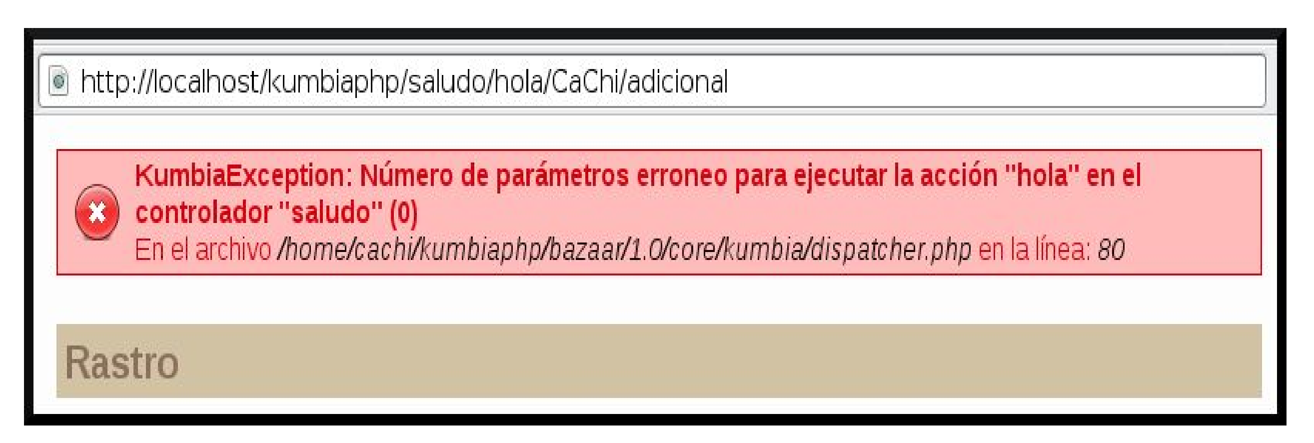


Figura 3.1: Excepción de Parámetros erróneos.

Siguiendo en el mismo ejemplo imaginemos que requerimos que la ejecución de la acción *hola()* obvie la cantidad de parámetros enviados por URL, para esto solo tenemos que indicarle a KumbiaPHP mediante el atributo *$limit_params* que descarte el número de parámetros que se pasan por URL.

```php
<?php
/**
 * Controller Saludo
 */
class SaludoController extends AppController {
    /**
     * Limita la cantidad correcta de
     * parámetros de una action
     */
    public $limit_params = FALSE;
    ... métodos ...
}
```

Cuando tiene el valor FALSE como se explicó antes, descarta la cantidad de parámetros de la acción. Ingresa a la siguiente URL http://localhost/kumbiaphp/saludo/hola/CaChi/param2/param3/ y verá como ya no esta la excepción de la figura 3.1 y podrá ver la vista de la acción como muestra la figura 3.2.



Figura 3.2: Descartando la cantidad de parámetros de la acción.

3.3. Convenciones y Creación de un Controlador

3.3.1. Convenciones

Los controladores en KumbiaPHP deben llevar las siguientes convenciones y características:
El archivo debe creado sólo en el directorio *app/controllers/*. El archivo debe tener el nombre del controlador y la terminación *_controller.php*, por ejemplo *saludo_controller.php*.

El archivo debe contener la clase controladora con el mismo nombre del archivo en notación CamelCase. Retomando el ejemplo anterior el nombre de la clase controladora sería SaludoController.
*/
public $limit_params = FALSE;
... métodos ...

}

3.3.2. Creación de un Controlador

Ahora se pondrá en práctica lo visto anteriormente y crearemos un controlador (controller) llamado saludo.

```
<?php
/**
 * Controller Saludo
 */
class SaludoController extends AppController {
}
```

3.3.3. Clase AppController

Es importante recordar que KumbiaPHP es un framework MVC y POO. En este sentido existe *AppController* y es la super clase de los controladores, todos deben heredar (extends) de esta clase para tener las propiedades (atributos) y métodos que facilitan la interacción entre la capa del modelo y presentación.

La clase *AppController* está definida en *app/libs/app_controller.php* es una clase muy sencilla de usar y es clave dentro del MVC.

3.3.4. Acciones y Controladores por defecto

3.4. Filtros

Los controladores en KumbiaPHP poseen unos métodos útiles que permiten realizar comprobaciones antes y después de ejecutar un controlador y una acción, los filtros pueden ser entendido como un mecanismo de seguridad en los cuales se puede cambiar el procesamiento de la petición según se requiera (por ejemplo verificar si un usuarios se encuentra autenticado en el sistema).

KumbiaPHP corre los filtros en un orden lógico, para manipular comprobaciones, a nivel de toda la aplicación o bien en particularidades de un controlador.

3.4.1. Filtros de Controladores

Los filtros de controladores se ejecutan antes y después de un controlador son útiles para comprobaciones a nivel de aplicación, como por ejemplo verificar el módulo que se esta intentando acceder, sesiones de usuarios, etc. Igualmente se puede usar para proteger nuestro controlador de información inadecuada.

Los filtros son métodos los cuales sobreescribimos (característica POO) para darle el

comportamiento deseado.

3.4.1.1. initialize()

KumbiaPHP llama al método *initialize()* antes de ejecutar el controlador y se encuentra definido para ser usado en la clase AppController (ver sección 3.3.3).

3.4.1.2. finalize()

KumbiaPHP llama al método *finalize()* después de ejecutar el controlador y se encuentra definido para ser usado en la clase AppController (ver sección 3.3.3).

3.4.2. Filtros de Acciones

Los filtros de acciones se ejecutan antes y después de una acción son útiles para comprobaciones a nivel de controller, como por ejemplo verificar que una petición es asíncrona, cambiar tipos de respuesta, etc. Igualmente se puede usar para proteger nuestra acción de información inadecuada que sea enviada a ellos.

3.4.2.1. before_filter()

KumbiaPHP llama al método *before_filter()* antes de ejecutar la acción del controlador y es útil para verificar si una petición es asíncrona entre otros.

3.4.2.2. after_filter()

KumbiaPHP llama al método *after_filter()* después de ejecutar la acción del controlador y es útil para cambiar valores de sesión entre otros.

4 La Vista

KumbiaPHP posee un sistema de presentación basado en Vistas (Views) que viene siendo el tercer componente del sistema MVC como se vió en la sección “Modelo, Vista, Controlador”, en este sentido las vistas son plantillas de código reutilizable que sirven para mostrar los datos al usuario y se encuentran ubicadas en el directorio *app/views/*.

Es buena práctica de desarrollo que las vistas contengan una cantidad mínima de código en PHP para que sea suficientemente entendible para un diseñador Web y además, para dejar a las vistas solo las tareas de visualizar los resultados generados por los controladores y presentar las capturas de datos para usuarios.

El manejador de vistas implementa el patrón de diseño de vista en dos pasos, el cual consiste en dividir el proceso de mostrar una vista en dos partes: la primera parte es utilizar una vista o «view» asociada a una acción del controlador para convertir los datos que vienen del modelo en lógica de presentación sin especificar ningún formato específico y la segunda es establecer el formato de presentación a través de una plantilla o «template».

Asimismo tanto las vistas de acción como las plantillas pueden utilizar vistas parciales o «partials». Estas vistas parciales son fragmentos de vistas que son compartidas por distintas vistas, de manera que constituyen lógica de presentación reutilizable en la aplicación. Ejemplos: menús, cabeceras, pies de página, entre otros.

KumbiaPHP favoreciendo siempre los convenios asume los siguientes respecto a las vistas:

- Todos los archivos de vistas deben tener la extensión *.phtml*.
- Cada controlador tiene un directorio de vistas asociado cuyo nombre coincide con el nombre del controlador en notación smallcase. Por ejemplo: si posees un controlador cuya clase se denomina «PersonalTecnicoController» esta por convenio tiene un directorio de vistas «personal_tecnico».
- Cada vez que se ejecuta una acción se intenta cargar una vista cuyo nombre es el mismo que el de la acción ejecutada.
- Los templates deben ubicarse en el directorio *views/_shared/templates*.
- Los partials deben ubicarse en el directorio *views/_shared/partials*.
- Por defecto se utiliza el template «default» para mostrar las vistas de acción.

Para indicar una vista diferente a la asumida por convención se debe utilizar el método *View::select()* en el controlador. Por ejemplo:

```
<?php
class SaludoController extends AppController {
    public function saludo(){
       View::select('hola');
```

```
    }
}
```

De esta manera luego de que se ejecute la acción «saludo» se mostrará la vista *saludo/hola.phtml* utilizando el template *default*.

En caso de que no desee mostrar una vista, solamente debe pasar *NULL* como argumento de *View::select()*.

```
<?php
class SaludoController extends AppController {
    public function index(){
       View::select(NULL);
    }
}
```

Para finalizar este apartado cabe destacar que tanto las vistas de acción, los templates y los partials son vistas, pero por comodidad se suele referir a la vista de acción sencillamente bajo el nombre de «vista».

4.1 Pasando datos a la vista

Para pasar datos a la vista estos deben cargarse como atributos públicos del controlador y luego de que se ejecute la acción, el manejador de vistas cargará los atributos públicos del controlador como variables de ámbito local en la vista. Ejemplo:

El controlador: *controllers/saludo_controller.php*

```
<?php
class SaludoController extends AppController{
    public function hola(){
       $this->usuario = 'Mundo';
    }
}
```

La vista: *views/saludo/hola.phtml*

```
Hola <?php echo $usuario ?>
```

4.2 Buffer de salida

Para mostrar el contenido del buffer de salida se hace uso del método *View::content()*, donde el contenido del buffer de salida lo constituye principalmente los echo o print que efectué el usuario y asimismo los mensajes Flash. Al invocar *View::content()* se muestra el contenido del buffer de salida en el lugar donde fue invocado.

El controlador: *saludo_controller.php*

```
<?php
class SaludoController extends AppController {
    public function hola(){
       Flash::valid('Hola Mundo');
    }
}
```

La vista: *hola.phtml*

```
Saludo realizado:
<?php View::content() ?>
```

4.3 Template

Los templates constituyen la capa más externa de la vista que se mostrará luego de ejecutar una acción del controlador, de manera que permite establecer el formato de presentación apropiado para la vista.

Cuando se habla de formato no se refiere únicamente al tipo de documento, si no también a elementos como cabeceras y menus. Por ende el template está compuesto por aquellos elementos que en conjunto son utilizados para la presentación de diversas vistas, dando de esta manera un formato de presentación reutilizable.

4.3.1 ¿Como crear un Template?

Para construir un nuevo template se debe crear un archivo con extensión .phtml en el directorio *views/_shared/templates/* el cual debe corresponder con el nombre del template.

Como se explicó anteriormente al inicio del capítulo "La Vista", el manejador de vistas utiliza el patrón de diseño de «vista en dos pasos». En el primer paso, se procesa la vista de acción, luego la vista de acción procesada se almacena en el buffer de salida y en el segundo paso se procesa el template.

En consecuencia, como la vista de acción procesada se acumula en el buffer de salida es necesario invocar el método *View::content()* en el lugar donde se desea mostrar la vista, tal como se indicó en la sección 4.2.

Ejemplo:

views/_shared/templates/ejemplo.phtml

```
<!DOCTYPE html>
<html>
<head>
    <title>Template de Ejemplo</title>
</head>
<body>
    <h1>Template de Ejemplo</h1>

    <?php View::content() ?>
</body>
</html>
```

4.3.2 ¿Como utilizar un template?

Para seleccionar el template a utilizar se debe invocar el método *View::template()* pasando como argumento el template a utilizar. Ejemplo:

En el controlador:

```
<?php
class SaludoController extends AppController {
    public function hola(){
       // Selecciona el template 'mi_template.phtml'
       View::template('mi_template');
    }
}
```

Asimismo es posible indicar al manejador de vistas que no utilice ningún template y por lo tanto muestre solamente la vista, para esto se debe pasar NULL como argumento a *View::template()*.

```
<?php
class SaludoController extends AppController {
    public function hola(){
       // No utilizar template
       View::template(NULL);
    }
}
```

4.3.3 Pasando datos al template

Como se vio en la sección “Pasando datos a la vista”, los atributos públicos del controlador se cargan como variables de ámbito local en la vista, como mostrar el template, constituye el

segundo paso para mostrar la vista completa, los atributos públicos del controlador estarán de igual manera cargados como variables de ámbito local en el template. Ejemplo:

En el controlador *saludo_controller.php*

```
<?php
class SaludoController extends AppController {
    public function hola(){
       Flash::valid('Hola Mundo');

       // Pasando el titulo para la página
       $this->titulo = 'Saludando al Mundo';

       /* No se utilizará vista, por lo tanto la
             salida será del buffer y template */
       View::select(NULL, 'saludo');
    }
}
```

En el template *saludo.phtml*

```
<!DOCTYPE html>
<html>
<head>
    <title><?php echo $titulo ?></title>
</head>
<body>
    <h1>Template de Saludo</h1>

    <?php View::content() ?>
</body>
</html>
```

4.4 Partial

Los partials o «vistas parciales» son fragmentos de vistas que son compartidas por distintas vistas, de manera que constituyen lógica de presentación reutilizable en la aplicación. Por lo general los partials son elementos como: menús, cabecera, pie de página, formularios, entre otros.

4.4.1 ¿Cómo crear un partial?

Para construir un nuevo partial se debe crear un archivo con extensión .phtml en el directorio

views/_shared/partials/ el cual debe corresponder con el nombre del partial.

Ejemplo:

views/_shared/partials/cabecera.phtml

```
<h1>Template de Saludo</h1>
```

4.4.2 ¿Cómo utilizar un partial?

Para utilizar un partial se debe invocar el método *View::partial()* indicando como argumento el partial deseado y la vista parcial se mostrará en el lugar donde se invocó.

Ejemplo utilizando un partial en un template:

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo</title>
</head>
<body>
    <?php View::partial('cabecera') ?>

    <?php View::content() ?>
</body>
</html>
```

Cabe destacar que los partial se pueden utilizar tanto en vistas de acción, templates e incluso dentro de otros partials.

4.4.3 Pasando datos a los partials

Para pasar datos a un partial, estos se deben indicar en un array asociativo donde cada clave con su correspondiente valor se cargarán como variables en el ámbito local del partial.

Ejemplo:

views/partials/cabecera.phtml

```
<h1>Título: <?php echo $titulo ?></h1>
```

views/ejemplo/index.phtml

```
<?php View::partial('cabecera', FALSE, array('titulo' => 'Ejemplo'))
```

```
?>
<p>
Este es un ejemplo
</p>
```

4.5 Agrupando en directorios

En KumbiaPHP tanto las vistas, los partials y los templates pueden agruparse en directorios, utilizando el separador «/» en la ruta .

4.5.1 Ejemplo de agrupación de vista

La vista *views/usuario/clasificado/fomulario.phtml*, se utiliza de la siguiente manera en el controlador:

```
<?php
class UsuarioController extends AppController {
    public function nuevo(){
       // Selecciona la vista
       View::select('clasificado/formulario');
    }
}
```

4.5.2 Ejemplo de agrupación de partial

El partial *views/_shared/partials/usuario/formulario.phtml*, se utiliza de la siguiente manera ya sea en vista o en template:

```
<h1>Nuevo Usuario</h1>
<?php View::partial('usuario/formulario') ?>
```

4.5.3 Ejemplo de agrupación de template

El template *views/_shared/templates/usuario/administrador.phtml*, se utiliza de la siguiente manera en el controlador:

```php
<?php
class AdministradorController  extends AppController {
    protected function before_filter(){
      // Selecciona el template
      View::template('usuario/administrador');
    }
}
```

4.6 Tipos de respuestas

Los tipos de respuestas son utilizados para establecer distintos formatos de la vista. Por ejemplo: xml, json y pdf.

Para establecer un tipo de respuesta se debe invocar el método *View::response()* indicando la respuesta deseada, una vez que se indica el tipo de respuesta este es automáticamente colocado como extensión del archivo de vista. En consecuencia utilizar los tipos de respuestas en conjunto a los template constituyen una potente herramienta para generación de vistas completas para el usuario.

Ejemplo:

```php
<?php
class UsuarioController  extends AppController {
    public function index(){
      // Establece el tipo de respuesta
      View::response('json');
    }
}
```

En este ejemplo se mostrará la vista *index.json.phtml*.

4.7 Uso de cache en las vistas

El manejador de vistas proporciona mecanismos a través de los cuales las vistas, los partials y los templates se pueden cachear, el usuario indica el tiempo durante el cual estos estarán almacenados en la caché de manera que el manejador de vistas cargará estos elementos sin necesidad de procesarlos, aumentando el rendimiento de la aplicación.

En este sentido para indicar el tiempo de cache se sigue el formato de la función strtotime de PHP. Ejemplo: '+1 week';

4.7.1 Cache de vistas

Para cachear una vista se utiliza el método *View::cache()* en el controlador.

```
<?php
class UsuarioController extends AppController {
    public function index(){
       // Indica el tiempo de cache de la vista
       View::cache('+20 days');
    }
}
```

Cabe destacar que la acción en el controlador se ejecuta, debido a que los datos pasados a la vista pueden de igual manera ser requeridos en el template.

4.7.1.1 Grupos de vistas en cache

Las vistas cacheadas se pueden almacenar en grupos. Los grupos son muy interesantes, ya que se puede borrar la cache por grupos también. Ejemplo: guardar caché de posts en un grupo, al crear, editar o borrar un post, podemos borrar la caché de ese grupo, para que se regenere la cache.

En este caso es necesario indicar en el método *View::cache()* que se cacheará una vista en un grupo específico.

```
<?php
class UsuarioController extends AppController {
    public function index(){
       // Indica el tiempo de cache de la vista
       View::cache('+20 days', 'view', 'miGrupo');
    }
}
```

4.7.2 Caché de templates

Cachear un template consiste en cachear en conjunto tanto la vista y template para una url específica. Para cachear un template se usa el método *View::cache()* en el controlador indicando el tiempo durante el cual estará cacheado el template.

```
<?php
```

```
class UsuarioController extends AppController {
    public function index(){
      // Indica el tiempo de cache de template
      View::cache('+20 days', 'template');
    }
}
```

Cabe destacar que para aumentar el rendimiento no se ejecuta la acción en el controlador, debido a que mostrar el template es el último paso que realiza el manejador de vistas para mostrar la vista al usuario y en este paso ya todos los datos enviados a la vista y template han sido utilizados.

4.7.3 Cache de partials

Para cachear partials se debe indicar como segundo argumento al invocar *View::partial()* el tiempo durante el cual se cacheará.

```
<?php View::partial('usuario', '+1 day') ?>
```

```
<?php View::partial('usuario', '+1 day', array('nombre' => 'pepe'))
?>
```

4.8 Helpers

Los helpers (ayudas) se usan en los views. Encapsulan código en métodos para su fácil reuso. KumbiaPHP ya viene con helpers creados.
Pero lo realmente útil, es que los usuarios se pueden crear sus propios helpers y colocarlos en app/extensions/helpers/. Y después usarlos tranquilamente en sus views, KumbiaPHP se encarga de cargar transparentemente sus helpers así como los uses.

4.8.1 Clase Html

Clase con métodos estáticos con la que podemos crear etiquetas HTML optimizadas respetando las convenciones de KumbiaPHP.

Html::img()

Permite incluir una imagen

```
$src ruta de la imagen
$alt atributo alt para la imagen
$attrs atributos adicionales

img ($src, $alt=NULL, $attrs = NULL)
```

```
/*Ejemplo*/
echo Html::img('spin.gif','una imagen'); //se muestra la imagen
spin.gif que se encuentra dentro de "/public/img/"
//con el artibuto alt 'una imagen'
```

Html::link()

Permite incluir un link

```
$action ruta a la acción
$text texto a mostrar
$attrs atributos adicionales

Html::link ($action, $text, $attrs = NULL)
```

```
/*Ejemplo*/
echo Html::link('pages/show/kumbia/status','Configuración'); //se
muestra un link con el texto 'Configuración'
```

Html::lists()

Crea una lista html a partir de un array

```
$array contenido de la lista
$type por defecto ul, y si no ol
$attrs atributos adicionales

Html::lists($array, $type = 'ul', $attrs = NULL)
```

```
/*Ejemplo*/
$ar = array('Abdomen' => 'Abdomen',
            'Brazos' => 'Brazos',
            'Cabeza' => 'Cabeza',
            'Cuello' => 'Cuello',
```

```
            'Genitales' => 'Genitales',
            'Piernas' => 'Piernas',
            'Tórax' => 'Tórax',
            'Otros' => 'Otros');
//$ar el array que contiene los items de la lista
echo Html::lists($ar, $type = 'ol'); //Muestra una lista <ol></ol>

$ar2 =
array('Abdomen','Brazos','Cabeza','Cuello','Genitales','Piernas','Tó
rax','Otros');
echo Html::lists($ar2, $type = 'ol'); //Muestra una lista <ol></ol>
```

Html::gravatar()

Incluye imágenes de gravatar.com

```
$email Correo para conseguir su gravatar
$alt Texto alternativo de la imagen. Por defecto: gravatar
$size Tamaño del gravatar. Un numero de 1 a 512. Por defecto: 40
$default URL gravatar por defecto si no existe, o un default de
gravatar. Por defecto: mm

Html::gravatar($email, $alt='gravatar', $size=40, $default='mm')
```

```
echo Html::gravatar( $email ); // Simple
echo Html::link( Html::gravatar($email), $url); // Un gravatar que
es un link
echo Html::gravatar( $email, $name, 20,
'http://www.example.com/default.jpg '); //Completo
```

Html::includeCss()

Incluye los archivos CSS que previamente fueron cargados a la lista mediante Tag::css()

```
Tag::css('bienvenida');     //Pone en lista un CSS
(app/public/css/bienvenida.css)
echo Html::includeCss();  //Adiciona los recursos enlazados de la
clase en el proyecto
```

Html::meta()

Crea un metatag y lo agrega a una lista estática que será añadida más adelnate mediante

Html::includeMetatags();

```
$content contenido del metatag
$attrs atributos adicionales del tag

Html::meta($content, $attrs = NULL)
```

```
Html::meta('Kumbiaphp-team',"name = 'Author'");
//Agrega: <meta content="Kumbiaphp-team" name = 'Author' />
Html::meta('text/html; charset=UTF-8',"http-equiv =
'Content-type'");
//Agrega: <meta content="text/html; charset=UTF-8" http-equiv =
'Content-type' />
```

Html::includeMetatags()

Agrega los metatag que previamente se habían agregado

```
Html::meta('Kumbiaphp-team',"name = 'Author'");
Html::meta('text/html; charset=UTF-8',"http-equiv =
'Content-type'");
echo Html::includeMetatags(); //Visualiza <meta
content="Kumbiaphp-team" name = 'Author'/>
```

Html::headLink()

Agrega un elemento de vínculo externo de tipo <link> a la cola de enlaces (para poder ser visualizado se requiere de Html::includeHeadLinks() de modo similar que Html::includeCss())

```
$href dirección url del recurso a enlazar
$attrs atributos adicionales

Html::headLink($href, $attrs = NULL)
```

```
Html::headlink('http://www.kumbiaphp.com/public/style.css',"rel='sty
lesheet',type='text/css' media='screen'"); //Se agrega a la cola de
links el enlace a un recurso externo, en este caso la hoja de estilo
ubicada en "http://www.kumbiaphp.com/public/style.css"

/*Agrega a la cola de links "<link rel="alternate"
type="application/rss+xml" title="KumbiaPHP Framework RSS Feed"
href="http://www.kumbiaphp.com/blog/feed/" />" con lo cual podemos
```

```
incluir un feed sin usar las convenciones de kumbiaphp */

Html::headlink('http://www.kumbiaphp.com/blog/feed/ ',"rel='alternate
' type='application/rss+xml' title='KumbiaPHP Framework RSS Feed'" );
Html::headlink('http://www.kumbiaphp.com/favicon.ico ',"rel='shortcut
icon',type='image/x-icon'" ); //Agrega la etiqueta <link> para usar un
favicon externo

echo Html::includeHeadLinks(); //Muestra los links que contiene la
cola
```

Html::headLinkAction()

Agrega un elemento de vínculo interno de tipo <link> a la cola de enlaces (para poder ser visualizado se requiere de Html::includeHeadLinks() de modo similar que Html::includeCss()) respetando las convenciones de KumbiaPHP.

```
$href dirección url del recurso a enlazar
$attrs atributos adicionales

Html::headLinkAction($action, $attrs = NULL)
```

```
/*Agrega a la cola de links "<link rel="alternate"
type="application/rss+xml" title="KumbiaPHP Framework RSS Feed"
href="http://www.kumbiaphp.com/blog/feed/" />" con lo cual podemos
incluir un feed usando las convenciones de KumbiaPHP.
Siendo 'articulos/feed' el nombre de la vista con el contenido del
feed */

Html::headLinkAction('articulos/feed', "rel='alternate'
type='application/rss+xml' title='KumbiaPHP Framework RSS Feed'" );

echo Html::includeHeadLinks(); //Muestra los links que contiene la
cola
```

Html::headLinkResource()

Agrega un elemento de vínculo a un recurso interno con la etiqueta <link> a la cola de enlaces (para poder ser visualizado se requiere de Html::includeHeadLinks())

```
$resource ubicación del recurso en public
$attrs atributos adicionales
```

```
Html::headLinkResource($resource, $attrs = NULL)
```

```
Html::headLinkResource('favicon.ico',"rel='shortcut
icon',type='image/x-icon'"); //Agrega la etiqueta <link> para usar un
favicon interno ubicado en el directorio '/public/'

echo Html::includeHeadLinks(); //Muestra los links que contiene la
cola
```

Html::includeHeadLinks()

Incluye los links que previamente se pusieron en cola

```
Html::headlink('http://www.kumbiaphp.com/favicon.ico',"rel='shortcut
icon',type='image/x-icon'"); //Agrega la etiqueta <link> para usar un
favicon externo
Html::headLinkAction('articulos/feed', "rel='alternate'
type='application/rss+xml' title='KumbiaPHP Framework RSS Feed'" );
echo Html::includeHeadLinks();
```

4.8.2. Clase Tag

Esta clase nos va a permitir adicionar archivos JS y CSS a nuestro proyecto, bien sean archivos que se encuentren en nuestro servidor o en un servidor externo. También vamos a poder hacer

Las funciones de esta clase son de tipo estáticas, lo que nos permite usarlas directamente de la forma como se presentan a continuación.

Tag::css()

Incluye un archivo CSS a la lista

```
Tag::css('bienvenida');    //Pone en lista un CSS
(app/public/css/bienvenida.css)
echo Html::includeCss();  //Adiciona los recursos enlazados de la
clase en el proyecto
```

Tag::js()

Incluye un archivo JavaScript a la vista, partial o template

```
<?php
echo Tag::js('jquery/jquery.kumbiaphp' ); //Adiciona un archivo
javascript (/app/public/javascript/jquery/jquery.kumbiaphp.js)
?>
```

4.8.3. Clase Form

Clase para el manejo y la creación de formularios

Form::open()

Crea una etiqueta de formulario

```
$action acción a  la que envía los datos, por defecto llama la
misma acción de donde proviene
$method 'POST', 'GET', 'DELETE', 'HEAD', 'PUT'. Por defecto se
tiene en 'POST'
$attrs atributos adicionales

Form::open($action = NULL, $method = 'POST', $attrs = NULL)
```

```
/*Ejemplo*/
<?php echo Form::open(); ?>//inicia un formulario que enviara los
datos a la acción que corresponde al controller actual
<?php echo Form::open('usuarios/nuevo'); ?> //inicia un formulario
que enviara los datos al controller 'usuarios' y la acción 'nuevo'
```

Form::openMultipart()

Crea una etiqueta de formulario multipart, este es ideal para formularios que contienen campos de subida de archivos

```
$action acción a  la que envía los datos, por defecto llama la
misma acción de donde proviene
$attrs atributos adicionales

Form::openMultipart ($action = NULL, $attrs = NULL)
```

```
/*Ejemplo*/
echo Form::openMultipart();//inicia un formulario multipart que
enviara los datos a la acción que corresponde a la vista actual
echo Form::openMultipart('usuarios/nuevo');//inicia un formulario
```

```
multipart que enviara los datos al controller 'usuario' y la acción
'nuevo'
```

Form::close()

Crea una etiqueta de cierre de formulario

```
/*Ejemplo*/
echo Form::close();
//crea una etiqueta de cierre de formulario </form>
```

Form::input()

Crea un campo de tipo input

```
$attrs atributos para el tag
$content contenido interno

Form::input($attrs = NULL, $content = NULL)
```

```
/*Ejemplo*/
echo Form::input('nombre');
```

Form::text()

Crea un campo de tipo input

Siempre que se le da el parámetro name de la forma model.campo, es decir un nombre que contenga un punto dentro del string, se crea el campo de texto con el name= "model[campo]" y el id="model_campo".

```
$field Nombre de campo
$attrs atributos de campo
$value valor inicial para el input

Form::text($field, $attrs = NULL, $value = NULL)
```

```
/*Ejemplo*/
echo Form::text('nombre'); //crea un campo de tipo texto con el
parámetro name= "nombre", id = "nombre"
echo Form::text('usuario.nombre'); //crea un campo de tipo texto con
```

```
el parámetro name= "usuario[nombre]", id = "usuario.nombre"
echo Form::text('nombre',"class= 'caja'",'55'); //crea un campo de
tipo texto con el parámetro name= "nombre", id = "nombre", class=
"caja", value = "55"
```

Form::pass()

Crea un campo de tipo Password

```
$field nombre de campo
$attrs atributos de campo
$value valor inicial para el campo

Form::pass($field, $attrs = NULL, $value = NULL)
```

```
/*Ejemplo*/
echo Form::pass('password'); //crea un campo de tipo password con el
parámetro name= "password"
```

Form::textarea()

Crea un textarea

```
$field nombre de campo
$attrs atributos de campo
$value valor inicial para el textarea

Form::textarea($field, $attrs = NULL, $value = NULL)
```

```
echo Form::textarea('detalles'); //Crea un textarea
```

Form::label()

Crea un label y lo asocia a un campo

```
$text texto a mostrar
$field campo al que hace referencia
$attrs array de atributos opcionales

Form::label($text, $field, $attrs = NULL)
```

```
echo Form::label('nombre de usuario:' ,'nombre'); //Crea un label para el campo nombre con el texto 'nombre de usuario:'
echo Form::text('nombre');
```

Form::hidden()

Crea un campo hidden (campo oculto)

```
$field nombre de campo
$attrs atributos adicionales de campo
$value valor inicial para el campo oculto

Form::hidden($field, $attrs = NULL, $value = NULL)
```

```
echo Form::hidden( 'id', NULL, 12); //Crea un campo oculto con el name="id" y el value="12"
```

Form::dbSelect()

Crea campo Select que toma los valores de objetos de ActiveRecord, para esta versión del framework el uso de este helper ha sido simplificado. Ya no es necesario instanciar el modelo.

```
$field nombre del modelo y campo pk (bajo la convención modelo.campo_id)
 $show campo que se mostrará
 $data array de valores, array('modelo','método','param')
$blank campo en blanco
$attrs atributos de campo
$value valor inicial para el campo


Form::dbSelect($field, $show = NULL, $data = NULL, $blank = NULL, $attrs = NULL, $value = NULL)
```

Vista

```
echo Form::dbSelect('usuarios.campo_id'); //la forma más fácil, carga el modelo(campo) y muestra el primer campo después del pk(id)
echo Form::dbSelect('usuarios.campo_id', 'campo'); //muestra el campo y lo ordena ascendentemente
```

Form::select()

Crea un campo Select (un combobox)

```
$field nombre de campo
$data array de valores para la lista desplegable
$attrs atributos de campo
$value valor inicial para el campo

Form::select($field, $data, $attrs = NULL, $value = NULL)
```

```
$ar2 =
array('Abdomen','Brazos','Cabeza','Cuello','Genitales','Piernas','Tó
rax','Otros');
echo Form::Select('region', $ar2, NULL, 'Cuello'); //Crea un campo
Select (un combobox) con el nombre 'region' y teniendo preseleccionado
'Cuello'
```

Resultado:

```
<select id="region" name="region">
<option value="0">Abdomen</option>
<option value="1">Brazos</option>
[...]
</select>
```

Otra Posibilidad:

```
$ar2 =
array('Abdomen'=>'Abdomen','Brazos'=>'Brazos','Cabeza'=>'Cabeza','Cu
ello'=>'Cuello','Genitales'=>'Genitales','Piernas'=>'Piernas','Tórax
'=>'Tórax','Otros'=>'Otros');
echo Form::Select('region', $ar2, NULL, 'Cuello');
```

Resultado:

```
<select id="region" name="region">
<option value="Abdomen">Abdomen</option>
<option value="Brazos">Brazos</option>
[...]
</select>
```

Form::file()

Crea campo File para subir archivos, el formulario se debe abrir con Form::openMultipart()

```
$field nombre de campo
$attrs atributos de campo

Form::file($field, $attrs = NULL)
```

```
echo Form::openMultipart(); //Abre el formulario multipart
echo Form::file('subir'); crear el campo para subir archivos
echo Form::close(); //Cierra el formulario
```

Form::button()

Crea un botón

```
$text texto del botón
$attrs atributos del botón

Form::button($text, $attrs = NULL)
```

```
echo Form::button('calcular'); //Crea un botón con el texto
'calcular'
```

Form::submitImage()

Crea un botón de tipo imagen siguiendo las convenciones de KumbiaPHP, la imagen deberá estar dentro del directorio '/public/img/'

```
$img ruta de la imagen que usa el botón
$attrs atributos del botón

Form::submitImage($img, $attrs = NULL)
```

```
echo Form::submitImage('botones/edit.gif'); //Crea un botón con la
imagen 'botones/edit.gif'
```

Form::submit()

Crea un botón de submit para el formulario actual

```
$text texto del botón
$attrs atributos del botón
```

```
Form::submit($text, $attrs = NULL)
```

```
echo Form::submit('enviar'); //Crea un botón con el texto 'enviar'
```

Form::reset()

Crea un botón reset para el formulario actual

```
$text texto del botón
$attrs atributos del botón

Form::reset($text, $attrs = NULL)
```

```
echo Form::reset('reiniciar'); //Crea un botón con el texto
'reiniciar'
```

Form::check()

Crea un checkbox

```
$field nombre de campo
$value valor en el checkbox
$attrs atributos de campo
$checked indica si se marca el campo

Form::check($field, $value, $attrs = NULL, $checked = NULL)
```

```
echo Form::check('recuerdame','1','',true); // Crea un check
seleccionado con id="recuerdame" , name="recuerdame" y value="1"
echo Form::check('recuerdame','1','',false); // Crea un check NO
seleccionado con id="recuerdame" , name="recuerdame" y value="1"
```

Form::radio()

Crea un radio button

```
$field nombre de campo
$value valor en el radio
$attrs atributos de campo
```

```
$checked indica si se marca el campo

Form::radio($field, $value, $attrs = NULL, $checked = NULL)
```

```
$on = 'masculino';
echo Form::radio("rdo", 'masculino', NULL, TRUE);  //<input
id="rdo1" name="rdo" type="radio" value="masculino" checked="checked">
echo Form::radio("rdo", 'femenino'); //<input id="rdo2" name="rdo"
type="radio" value="femenino">
```

Js

Este helper ofrece algunas implementaciones que utilizan javascript simple.

Js::link ()

Crea un enlace que al pulsar muestra un dialogo de confirmación para redireccionamiento a la ruta indicada.

```
$action ruta a la accion
$text texto a mostrar
$confirm mensaje de confirmacion
$class clases adicionales para el link
$attrs $attrs atributos adicionales

Js::link ($action, $text, $confirm = '¿Está Seguro?', $class = NULL, $attrs = NULL)
```

```
<?php echo Js::link('usuario/eliminar/5', 'Eliminar'); ?>
```

Si desea aplicar una clase de estilo al enlace debe indicarlo en el argumento *$class*.

```
<?php echo Js::link('usuario/eliminar/5', 'Eliminar', '¿Está seguro de esta operación?',
'b_eliminar') ?>
```

Js::linkAction ()

Crea un enlace que al pulsar muestra un dialogo de confirmación para redireccionamiento a la acción indicada.

```
$action accion de controlador
$text texto a mostrar
$confirm mensaje de confirmacion
```

```
$class clases adicionales para el link
$attrs $attrs atributos adicionales

Js::linkAction($action, $text, $confirm = '¿Está Seguro?', $class = NULL, $attrs = NULL)
```

```
<?php echo Js::linkAction('eliminar/5', 'Eliminar'); ?>
```

Si desea aplicar una clase de estilo al enlace debe indicarlo en el argumento *$class*.

```
<?php echo Js::linkAction('eliminar/5', 'Eliminar', '¿Está seguro de esta operación?',
'b_eliminar') ?>
```

Js::submit ()

Crea un botón submit que al pulsar muestra un diálogo de confirmación.

```
$text texto a mostrar
$confirm mensaje de confirmacion
$class clases adicionales para el link
$attrs atributos adicionales

Js::submit ($text, $confirm = '¿Está Seguro?', $class = NULL, $attrs = NULL)
```

```
<?php echo Js::submit('Guardar') ?>
```

Si desea aplicar una clase de estilo al botón debe indicarlo en el argumento *$class*.

```
<?php echo Js::submit('Guardar', '¿Está Seguro?', 'boton_guardar') ?>
```

Js::submitImage ()

Crea un botón tipo image que al pulsar muestra un dialogo de confirmación.

```
$img ruta a la imagen
$confirm mensaje de confirmacion
$class clases adicionales para el link
$attrs atributos adicionales

Js::submitImage($img $confirm = '¿Está Seguro?', $class = NULL, $attrs = NULL)
```

```
<?php echo Js::submitImage(‘botones/guardar.png’) ?>
```

Si desea aplicar una clase de estilo al botón debe indicarlo en el argumento *$class*.

```
<?php echo Js::submitImage(‘botones/guardar’, ‘¿Está Seguro?’, ‘boton_guardar’) ?>
```

Ajax

Este helper ofrece implementaciones para facilitar la integración con AJAX.

Ajax::link()

Crea un enlace que actualiza la capa indicada con el contenido producto de la petición web.

```
$action ruta a la acción
$text texto a mostrar
$update capa a actualizar
$class clases adicionales
$attrs atributos adicionales

Ajax::link ($action, $text, $update, $class=NULL, $attrs=NULL)
```

Como ejemplo, crea un enlace que al pulsar emita un saludo. Con el fin anterior se tienen las siguientes vistas y controladores:

controllers/saludo_controller.php

```
<?php
class SaludoController extends AppController
{
    public function index()
    {}

    public function hola()
    {
        View::template(NULL);
    }
}
```

views/saludo/hola.phtml

```
Hola
```

views/saludo/index.phtml

```
<div id="capa_saludo"></div>
<?php
   echo Ajax::link('saludo/hola', 'Mostrar Saludo', 'capa_saludo');
   echo Tag::js('jquery/jquery+kumbiaphp.min');
?>
```

Al acceder a la acción *index* del controlador *saludo* se tiene:



Luego de pulsar el enlace se coloca el resultado de la petición ajax en la capa.



Ajax::linkAction()

Crea un enlace a una acción del controlador actual que actualiza la capa indicada con el contenido producto de la petición web.

```
$action accion
$text texto a mostrar
$update capa a actualizar
$class clases adicionales
$attrs atributos adicionales

Ajax::linkAction ($action, $text, $update, $class=NULL, $attrs=NULL)
```

```
<?php echo Ajax::linkAction('hola', 'Mostrar Saludo', 'capa_saludo') ?>
```

Por supuesto... aún falta a esta documentación, por el momento les recomiendo que revisen el CRUD de la versión 1.0 beta 2 allí podrán ver otros cambios, estos se documentaran muy pronto CRUD Beta2 KumbiaPHP

5 Modelos

En los Modelos reside la lógica de negocio (o de la aplicación). Equivocadamente, mucha gente cree que los modelos son sólo para trabajar con las bases de datos.

Los modelos pueden ser de muchos tipos:

- Crear miniaturas de imágenes
- Consumir y usar webservices
- Crear pasarelas Scaffold de pago
- Usar LDAP
- Enviar mails o consultar servidores de correo
- Interactuar con el sistema de ficheros local o vía FTP, o cualquier otro protocolo
- etc etc

5.1 ActiveRecord

5.2 Ejemplo sin ActiveRecord

5.3 Cómo usar los modelos

Los Modelos representan la lógica de la aplicación, y son parte fundamental para el momento que se desarrolla una aplicación, un buen uso de estos nos proporciona un gran poder al momento que se necesita escalar, mantener y reusar código en una aplicación.
Por lo general un mal uso de los modelos es solo dejar el archivo con la declaración de la clase y toda la lógica se genera en el controlador. Esta práctica trae como consecuencia que en primer lugar el controlador sea difícilmente entendible por alguien que intente agregar y/o modificar algo en esa funcionalidad, en segundo lugar lo poco que puedes rehusar el código en otros controladores y lo que hace es repetirse el código que hace lo mismo en otro controlador.
Partiendo de este principio los controladores **NO** deberían contener ningún tipo de lógica solo se encargan de atender las peticiones del usuario y solicitar dicha información a los modelos con esto garantizamos un buen uso del MVC.

5.1 El Modelo extiende el ActiveRecord

KumbiaPHP usa POO (Programación orientada a objetos), así que ActiveRecord es una clase que ya lleva métodos listos para usar. Estos métodos facilitan al usuario el manejo de las tablas de las bases de datos; entre ellos están los siguientes: find, find_all, save, update, etc.
El Modelo extiende la clase ActiveRecord para que el usuario pueda añadir sus propios

métodos, y así encapsular la lógica.

5.2 El Modelo extiende el ActiveRecord

KumbiaPHP usa POO (Programación orientada a objetos), así que ActiveRecord es una clase que ya lleva métodos listos para usar. Estos métodos facilitan al usuario el manejo de las tablas de las bases de datos; entre ellos están los siguientes: find, find_all, save, update, etc.

El Modelo extiende la clase ActiveRecord para que el usuario pueda añadir sus propios métodos, y así encapsular la lógica.

5.4 ActiveRecord API

A continuación veremos una referencia de los métodos que posee la clase ActiveRecord y su funcionalidad respectiva. Éstos se encuentran organizados alfabéticamente:

5.4.1 Consultas

Métodos para hacer consulta de registros:

5.4.1.1 distinct ()

Este método ejecuta una consulta de distinción única en la entidad, funciona igual que un "select unique campo" viéndolo desde la perspectiva del SQL. El objetivo es devolver un array con los valores únicos del campo especificado como parámetro.
Sintaxis

```
distinct([string $atributo_entidad], [ "conditions: …"], ["order: …"],
["limit: …"], ["column: …"])
```

Ejemplo

```
$unicos = Load::model('usuario')->distinct("estado")
# array('A', 'I', 'N')
```

Los parámetros conditions, order y limit funcionan idénticamente que en la función find y permiten modificar la forma o los mismos valores de retorno devueltos por ésta.

5.4.1.2 find_all_by_sql (string $sql)

Este método nos permite hacer una consulta por medio de un SQL y el resultado devuelto es un array de objetos de la misma clase con los valores de los registros en estos. La idea es que el uso de este método no debería ser común en nuestras aplicaciones ya que ActiveRecord se

encarga de eliminar el uso del SQL en gran porcentaje, pero hay momentos en que es necesario que seamos más específicos y tengamos que recurrir al uso de este.

Ejemplo

```
$usuarios = Load::model('usuario')->find_all_by_sql("select * from
usuarios where codigo not in (select codigo from ingreso)")
```

En este ejemplo consultamos todos los usuarios con una sentencia *where* especial. La idea es que los usuarios consultados no pueden estar en la entidad ingreso.

5.4.1.3 find_by_sql (string $sql)

Este método nos permite hacer una consulta por medio de un SQL y el resultado devuelto es un objeto que representa el resultado encontrado. La idea es que el uso de este método no debería ser común en nuestras aplicaciones ya que ActiveRecord se encarga de eliminar el uso del SQL en gran porcentaje, pero hay momentos en que es necesario que seamos más específicos y tengamos que recurrir al uso de este.

Ejemplo

```
$usuario = Load::model('usuario')->find_by_sql("select * from
usuarios where codigo not in (select codigo from ingreso) limit 1");
```

Este ejemplo consultamos todos los usuarios con una sentencia where especial e imprimimos sus nombres. La idea es que el usuario consultado no puede estar en la entidad ingreso.

5.4.1.4 find_first (string $sql)

Sintaxis

```
find_first([integer $id], ["conditions: …"], ["order: …"], ["limit:
…"], ["columns: …"])
```

El método “find_first” devuelve el primer registro de una entidad o la primera ocurrencia de acuerdo a unos criterios de búsqueda u ordenamiento. Los parámetros son todos opcionales y su orden no es relevante, cuando se invoca sin parámetros devuelve el primer registro insertado en la entidad. Este método es muy flexible y puede ser usado de muchas formas:

Ejemplo

```
$usuario = Load::model('usuario')->find_first("conditions: estado='A'
", "order: fecha desc");
```

En este ejemplo buscamos el primer registro cuyo estado sea igual a “A” y ordenado descendentemente, el resultado de éste, se carga a la variable $Usuarios e igualmente devuelve una instancia del mismo objeto ActiveRecord en caso de éxito o false en caso contrario.

Con el método find_first podemos buscar un registro en particular a partir de su id de esta forma:

```
$usuario = Load::model('usuario')->find_first(123);
```

Así obtenemos el registro 123 e igualmente devuelve una instancia del mismo objeto. ActiveRecord en caso de éxito o false en caso contrario. Kumbia genera una advertencia cuando los criterios de búsqueda para find_first devuelven más de un registro, para esto podemos forzar que se devuelva solamente uno, mediante el parámetro limit, de esta forma:

```
$usuario = Load::model('usuario')->find_first("conditions: estado='A'
","limit: 1");
```

Cuando queremos consultar sólo algunos de los atributos de la entidad podemos utilizar el parámetro *columns* así:

```
$usuario = Load::model('usuario')->find_first("columns: nombre,
estado");
```

Cuando especificamos el primer parámetro de tipo *string*, ActiveRecord asumirá que son las condiciones de búsqueda para find_first, así:

```
$usuario = Load::model('usuario')->find_first("estado=’A’");
```

De esta forma podemos también deducir que estas 2 sentencias arrojarían el mismo resultado:

```
$usuario = Load::model('usuario')->find_first("id=’123’");
```

```
$usuario = Load::model('usuario')->find_first(123);
```

5.4.1.5 find ()

Sintaxis

```
find([integer $id], ["conditions: …"], ["order: …"], ["limit: …],
["columns: …"])
```

El método "find" es el principal método de búsqueda de ActiveRecord, devuelve todas los registros de una entidad o el conjunto de ocurrencias de acuerdo a unos criterios de búsqueda. Los parámetros son todos opcionales y su orden no es relevante, incluso pueden ser combinados u omitidos si es necesario. Cuando se invoca sin parámetros devuelve todos los registros en la entidad.

No hay que olvidarse de incluir un espacio después de los dos puntos (:) en cada parámetro.

Ejemplo

```
$usuarios = Load::model('usuario')->find("conditions: estado='A'","order: fecha desc");
```

En este ejemplo buscamos todos los registros cuyo estado sea igual a "A" y devuelva éstos ordenados descendentemente, el resultado de este es un array de objetos de la misma clase con los valores de los registros cargados en ellos, en caso de no hayan registros devuelve un array vacío.

Con el método find podemos buscar un registro en particular a partir de su id de esta forma:

```
$usuario = Load::model('usuario')->find(123);
```

Así obtenemos el registro 123 e igualmente devuelve una instancia del mismo objeto ActiveRecord en caso de éxito o *false* en caso contrario. Como es un solo registro no devuelve un *array*, sino que los valores de éste se cargan en la misma variable si existe el registro.

Para limitar el número de registros devueltos, podemos usar el parámetro *limit*, así:

```
$usuarios = Load::model('usuario')->find("conditions: estado='A'",'limit: 5', 'offset: 1');
```

Cuando queremos consultar sólo algunos de los atributos de la entidad podemos utilizar el parámetro *columns* así:

```
$usuarios = Load::model('usuario')->find("columns: nombre, estado");
```

Cuando especificamos el primer parámetro de tipo *string*, ActiveRecord asumirá que son las condiciones de búsqueda para *find*, así:

```
$usuarios = Load::model('usuario')->find("estado='A'");
```

Se puede utilizar la propiedad *count* para saber cuántos registros fueron devueltos en la búsqueda.

Nota: No es necesario usar `find('id: $id')` para el find, se usa directamente `find($id)`

5.4.1.5 select_one (string $select_query)

Este método nos permite hacer ciertas consultas como ejecutar funciones en el motor de base de datos sabiendo que éstas devuelven un solo registro.

```
$current_time = Load::model('usuario')->select_one("current_time");
```

En el ejemplo queremos saber la hora actual del servidor devuelta desde MySQL así que podemos usar este método para esto.

5.4.1.6 select_one(string $select_query) (static)

Este método nos permite hacer ciertas consultas como ejecutar funciones en el motor de base de datos, sabiendo que éstas devuelven un solo registro. Este método se puede llamar de forma estática, esto significa que no es necesario que haya una instancia de ActiveRecord para hacer el llamado.

```
$current_time = ActiveRecord::select_one("current_time")
```

En el ejemplo queremos saber la hora actual del servidor devuelta desde MySQL así que podemos usar este método para esto.

5.4.1.7 exists()

Este método nos permite verificar si el registro existe o no en la base de datos mediante su id o una condición.

```
$usuario = Load::model('usuario');

$usuario->id = 3;

if($usuario->exists()){
    //El usuario con id igual a 3 si existe
}

Load::model('usuario')->exists("nombre='Juan Perez'")
Load::model('usuario')->exists(2); // Un Usuario con id->2?
```

5.4.1.8 find_all_by()

Este método nos permite realizar una búsqueda por algún campo

```
$resultado =
Load::model('producto')->find_all_by('categoria','Insumos');
```

5.4.1.9 find_by_*campo*()

Este método nos permite realizar una búsqueda por algún campo usando el nombre del método como nombre de éste. Devuelve un solo registro.

```
$resultado = Load:: model('producto')->find_by_categoria('Insumos');
```

5.4.1.10 find_all_by_*campo*()

Este método nos permite realizar una búsqueda por algún campo usando el nombre del método como nombre de éste. Devuelve todos los registros que coincidan con la búsqueda.

```
$resultado =
Load::model('producto')->find_all_by_categoria ("Insumos");
```

5.4.2 Conteos y sumatorias

5.4.2.1 count()

Realiza un conteo sobre los registros de la entidad con o sin alguna condición adicional. Emula la función de agrupamiento count.

```
$numero_registros = Load:: model('cliente')->count();
$numero_registros = Load:: model('cliente')->count("ciudad =
'BOGOTA'");
```

5.4.2.2 sum()

Realiza una sumatoria sobre los valores numéricos de el atributo de alguna entidad, emula la función de agrupamiento sum en el lenguaje SQL.

```
$suma = Load::model('producto')->sum("precio");
$suma = Load::model('producto')->sum("precio", "conditions: estado =
'A'");
```

5.4.2.3 count_by_sql()

Realiza una sumatoria utilizando lenguaje SQL.

```
$numero = Load::model('producto')->count_by_sql("select count(precio) from producto,
factura  where factura.codigo = 1124 \
 and factura.codigo_producto = producto.codigo_producto");
```

5.4.3 Promedios, máximo y mínimo

5.4.4 Creación, actualización y borrado de registros

5.4.5 Validaciones

5.4.6 Transacciones

5.4.7 Otros métodos

5.4.8 Callbacks ActiveRecord

5.4.9 Asociaciones

5.4.10 Paginadores

6 Scaffold

Introducción

Para empezar es importante saber, que el Scaffold se utilizó hasta la versión estable de Kumbiaphp 0.5 y que al salir la versión de Kumbiaphp 1.0 Spirit beta 1 se dejó a un lado, hasta crear uno nuevo más configurable y mantenible.

Viendo la necesidad y las facilidades que el Scaffold proporciona al apoyo de aplicaciones, el equipo de desarrollo de Kumbiaphp vuelve a incorporar un nuevo para su versión KumbiaPHP beta 2, mejorando y simplificando el desempeño del Scaffold para el Framework y que sin duda aporta a un gran avance en cualquier desarrollo de aplicación para usuarios iniciados en el uso de Kumbiaphp y usuarios avanzados, entregando para todos una gama alta de posibilidades.

Concepto

Scaffold es un método de meta-programación para construir aplicaciones de software que soportan bases de datos. Esta es una nueva técnica soportada por algunos frameworks del tipo MVC (Modelo-Vista-Controlador), donde el programador debe escribir una especificación que escriba como debe ser usada la aplicación de bases de datos. El compilador luego usará esta para generar un código que pueda usar la aplicación para leer, crear, actualizar y borrar entradas de la base de datos (algo conocido como CRUD o ABM), tratando de poner plantillas como un andamio Scaffold) en la cual construir una aplicación más potente.

Scaffolding es la evolución de códigos generadores de bases de datos desde ambientes más desarrollados, como ser CASE Generator de Oracle y otros tantos servidores 4GL para servicios al Cliente. Scaffolding se hizo popular gracias al framework "Ruby on Rails", que ha sido adaptado a otros frameworks, incluyendo Django, Monorail, KumbiaPHP framework entre otros.

Tomado de: Scaffolding Kumbiaphp

Objetivo

Crear un CRUD 100% Funcional con tan solo 2 líneas de código en mi controller.
KumbiaPHP tomará como por arte de magia, los parámetros indicados en mi TABLA y armará todo el CRUD.

Primeros Pasos

Para realizar nuestro primer Scaffold, vamos a utilizar el mismo modelo que trabajamos en el CRUD para KumbiaPHP Beta2, y que tiene por nombre menus.

Modelo

Crear el modelo, como de costumbre apuntando siempre a la clase ActiveRecord.

[app]/models/menus.php:

```
<?php
class Menus extends ActiveRecord{

}
```

Controlador

Crear el Controlador en este ejemplo, NO apuntaremos a la clase AppController y SI a la clase ScaffoldController.

[app]/controllers/menus_controller.php:

```
<?php
class MenusController extends ScaffoldController{
    public $model = 'menus';
}
```

Aquí terminan nuestros primeros pasos. No es necesario NADA MÁS. Tendremos por arte de magia un CRUD 100% Funcional.

Ventajas

1. Podremos ir cargando nuestros primeros registros en la BD
2. Pruebas al insertar registros
3. Avance progresivo, ya que podremos ir sustituyendo las vistas del Scaffold por mis propias vistas.

Desventaja

1. El Scaffold no es para hacer sistemas, si no para ayudar al principio de una aplicación.

Views para el scaffold

Por defecto usa los de views/_shared/scaffolds/kumbia/... Uno puede crear los suyos dentro de scaffolds views/_shared/scaffolds/foo/... y en el controller además del atributo $model añade; public $scaffold = 'foo';

Así usará los views de scaffolds/foo/...

Más importante es todavía, que uno puede crear sus views como siempre. es decir, si creas el controller MiController y creas el view en views/mi/editar.phtml (por ejemplo) usará primero el view, si no existe usará el de scaffolds. Así uno cambia los views a su gusto donde quiera y progresivamente.

7 Clases padre

7.1 AppController

7.2 ActiveRecord

Es la principal clase para la administración y funcionamiento de modelos. **ActiveRecord** es una implementación de este patrón de programación y está muy influenciada por la funcionalidad de su análoga en Ruby disponible en Rails. **ActiveRecord** proporciona la capa objeto-relacional que sigue rigurosamente el estándar ORM: Tablas en Clases, Registros en Objetos, y Campos en Atributos. Facilita el entendimiento del código asociado a base de datos y encapsula la lógica específica haciéndola más fácil de usar para el programador.

```
<?php
$cliente = Load::model('cliente');
$cliente ->nit = "808111827-2";
$cliente ->razon_social = "EMPRESA DE TELECOMUNICACIONES XYZ"
$cliente ->save();
```

7.2.1. Ventajas del ActiveRecord

- Se trabajan las entidades del Modelo más Naturalmente como objetos.
- Las acciones como Insertar, Consultar, Actualizar, Borrar, etc. de una entidad del Modelo están encapsuladas así que se reduce el código y se hace más fácil de mantener.
- Código más fácil de Entender y Mantener
- Reducción del uso del SQL en un 80%, con lo que se logra un alto porcentaje de independencia del motor de base de datos.
- Menos “detalles” más practicidad y utilidad
- ActiveRecord protege en un gran porcentaje de ataques de SQL inyection que puedan llegar a sufrir tus aplicaciones escapando caracteres que puedan facilitar estos ataques.

7.2.2. Crear un Modelo en Kumbia PHP Framework

Lo primero es crear un archivo en el directorio models con el mismo nombre de la relación en la base de datos. Por ejemplo: *models/clientes.php* Luego creamos una clase con el nombre de la tabla extendiendo alguna de las clases para modelos.

Ejemplo:

```
<?php
class Cliente extends ActiveRecord {
}
```

Si lo que se desea es crear un modelo de una clase que tiene nombre compuesto por ejemplo la clase **Tipo de Cliente**, por convención en nuestra base de datos esta tabla debe llamarse: **tipo_de_cliente** y el archivo: *models/tipo_de_cliente.php* y el código de ese modelo el siguiente:

```
<?php
class TipoDeCliente extends ActiveRecord {
}
```

7.2.3. Columnas y Atributos

Objetos **ActiveRecord** corresponden a registros en una tabla de una base de datos. Los objetos poseen atributos que corresponden a los campos en estas tablas. La clase **ActiveRecord** automáticamente obtiene la definición de los campos de las tablas y los convierte en atributos de la clase asociada. A esto es lo que nos referíamos con mapeo objeto relacional.

Miremos la tabla Álbum:

```
CREATE TABLE album (
	id INTEGER NOT NULL AUTO_INCREMENT,
	nombre VARCHAR(100) NOT NULL,
	fecha DATE NOT NULL,
	valor DECIMAL(12,2) NOT NULL,
	artista_id INTEGER NOT NULL,
	estado CHAR(1),
	PRIMARY KEY(id)
)
```

Podemos crear un **ActiveRecord** que mapee esta tabla:

```
<?php
class Album extends ActiveRecord {
}
```

Una instancia de esta clase será un objeto con los atributos de la tabla álbum:

```
<?php

$album = Load::model('album');
$album->id = 2;
$album->nombre = "Going Under";
$album->save();
```

O con...

```
<?php

Load::models('album');

$album = new Album();
$album->id = 2;
$album->nombre = "Going Under";
$album->save();
```

7.2.4. Llaves Primarias y el uso de IDs

En los ejemplos mostrados de KumbiaPHP siempre se trabaja una columna llamada id como llave primaria de nuestras tablas. Tal vez, esto no siempre es práctico a su parecer, de pronto al crear la tabla clientes la columna de número de identificación sería una excelente elección, pero en caso de cambiar este valor por otro tendría problemas con el dato que esté replicado en otras relaciones (ejemplo facturas), además de esto tendría que validar otras cosas relacionadas con su naturaleza. KumbiaPHP propone el uso de ids como llaves primarias con esto se automatiza muchas tareas de consulta y proporciona una forma de referirse unívocamente a un registro en especial sin depender de la naturaleza de un atributo específico. Usuarios de Rails se sentirán familiarizados con esta característica.

Esta particularidad también permite a KumbiaPHP entender el modelo entidad relación leyendo los nombres de los atributos de las tablas. Por ejemplo en la tabla álbum del ejemplo anterior la convención nos dice que id es la llave primaria de esta tabla pero además nos dice que hay una llave foránea a la tabla artista en su campo id.

7.2.5. Convenciones en ActiveRecord

ActiveRecord posee una serie de convenciones que le sirven para asumir distintas cualidades y relacionar un modelo de datos. Las convenciones son las siguientes:

id
Si ActiveRecord encuentra un campo llamado **id**, ActiveRecord asumirá que se trata de la llave primaria de la entidad y que es autonumérica.

tabla_id
Los campos terminados en **_id** indican relaciones foráneas a otras tablas, de esta forma se puede definir fácilmente las relaciones entre las entidades del modelo:

Un campo llamado **clientes_id** en una tabla indica que existe otra tabla llamada clientes y esta contiene un campo id que es foránea a este.

campo_at
Los campos terminados en **_at** indican que son fechas y posee la funcionalidad extra que obtienen el valor de fecha actual en una inserción

created_at es un campo fecha

campo_in
Los campos terminados en **_in** indican que son fechas y posee la funcionalidad extra que obtienen el valor de fecha actual en una actualización

modified_in es un campo fecha

NOTA: Los campos **_at** y **_in** deben ser de tipo fecha (date) en la RDBMS que se este utilizando.

View

...

8 Libs de KumbiaPHP

Kumbiaphp lleva clases listas para usar, pero recordad que podéis crearos vuestras propias clases para reutilizarlas en vuestros proyectos.También podéis usar clases externas a KumbiaPHP, como se explica en el próximo capítulo.

Caché

Un caché es un conjunto de datos duplicados de otros originales, con la propiedad de que los

datos originales son costosos de acceder, normalmente en tiempo, respecto a la copia en la caché.

El caché de datos está implementado en KumbiaPHP utilizando los patrones de diseño factory y singleton. Para hacer uso de la caché es necesario tener permisos de escritura en el directorio "cache" (solamente en el caso de los manejadores "sqlite" y "file").

Cada caché es controlada por un manejador de caché. El sistema de caché de KumbiaPHP actualmente posee los siguientes manejadores:

- **APC**: utiliza Alternative PHP Cache.
- **file**: caché en archivos, estos se almacenan en el directorio caché y compatible con todos los sistemas operativos.
- **nixfile**: caché en archivos, estos se almacenan en el directorio caché y compatible solo con sistemas operativos *nix (linux, freebsd, entre otros). Esta caché es más rápida que la caché «file».
- **sqlite**: caché utilizando base de datos SqLite y esta se ubica en el directorio caché.
- **memsqlite**: caché utilizando base de datos SqLite y los datos persisten en memoria durante la ejecución de la petición web.

Para obtener un manejador de caché se debe utilizar el método «driver» que proporciona la clase Cache.

driver($driver=null)

Este método permite obtener un manejador de caché específico (APC, file, nixfile, sqlite, memsqlite). Si no se indica, se obtiene el manejador de caché por defecto indicado en el *config.ini.*

```php
<?php
// cache por defecto
$data = Cache::driver()->get('data');

// manejador para memcache
$data_memcache = Cache::driver('memcache')->get('data');

// manejador para cache con APC
$data_apc = Cache::driver('APC')->get('data');
?>
```

get($id, $group='default')

Permite obtener un valor almacenado en la cache; es necesario especificar el parametro **$id** con el “id” correspondiente al valor en cache, tomando de manera predeterminada el grupo “default”.

save($value, $lifetime=null, $id=false, $group='default')

Permite guardar un valor en la caché, el tiempo de vida del valor en caché se debe especificar utilizando el formato de la función strtotime de php.

Al omitir parámetros al invocar el método save se comporta de la manera siguiente:

- Si no se especifica **$lifetime**, entonces se cachea por tiempo indefinido.
- Si no se especifica **$id** y **$group**, entonces se toma los indicados al invocar por última vez el método **get**.

```
<?php
$data = Cache::driver()->get('saludo');
if(!$data) {
   Cache::driver()->save('Hola', '+1 day');
}
echo $data;
?>
```

start ($lifetime, $id, $group='default')

Muestra buffer de salida cacheado, o en caso contrario inicia cacheo de buffer de salida hasta que se invoque el método end. Este método se utiliza frecuentemente para cachear un fragmento de vista.

```
<?php if(Cache::driver()->start('+1 day', 'saludo')): ?>
   Hola <?php echo $usuario ?>
   <?php Cache::driver()->end() ?>
<?php endif; ?>
```

end ($save=true)

Termina cacheo de buffer de salida indicando si se debe guardar o no en la caché.

Logger

La clase Logger para el manejo de Log fue reescrita de forma estática, esto quiere decir ya no es necesario crear una instancia de la clase Logger. Esta clase dispone de una variedad de métodos para manejar distintos tipos de Log.

```
<?php Logger::error('Mensaje de Error')?>
```

La salida de la instrucción anterior será lo siguiente:
[Thu, 05 Feb 09 15:19:39 -0500][ERROR] Mensaje de Error

Por defecto los archivos log tienen el siguiente nombre logDDMMYYY.txt este nombre puede ser cambiado si así lo deseamos a través de un parámetro adicional al método.

```
<?php Logger::error('Mensaje de Error', 'mi_log')?>
```

Se puede apreciar el segundo parámetro ahora el archivo tendrá como nombre mi_log.txt

Logger::warning ($msg);

Logger::error ($msg)

Logger::debug ($msg)

Logger::alert ($msg)

Logger::critical ($msg)

Logger::notice ($msg)

Logger::info ($msg)

Logger::emergence ($msg)

Logger::custom ($type='CUSTOM', $msg)

Flash

Flash es un helper muy útil en Kumbia que permite hacer la salida de mensajes de error, advertencia, informativos y éxito de forma estándar.

Flash::error($text)

Permite enviar un mensaje de error al usuario. Por defecto es un mensaje de letras color rojo y fondo color rosa pero estos pueden ser alterados en la clase css en public /css/style.css llamada error.

```
Flash::error(“Ha ocurrido un error”);
```

Flash::valid($text)

Permite enviar un mensaje de éxito al usuario. Por defecto es un mensaje de letras color verdes y fondo color verde pastel pero estos pueden ser alterados en la clase css en *public/css/style.css* llamada *valid*.

```
Flash::valid(“Se realizó el proceso correctamente”);
```

Flash::info($text)

Permite enviar un mensaje de información al usuario. Por defecto es un mensaje de letras color azules y fondo color azul pastel; pero estos pueden ser alterados en la clase css en *public/css/style.css* llamada *info*.

```
Flash::info(“No hay resultados en la búsqueda”);
```

Flash::warning($text)

Permite enviar un mensaje de advertencia al usuario. Por defecto es un mensaje de letras color azules y fondo color azul pastel pero estos pueden ser alterados en la clase css en *public/css/style.css* llamada *warning*.

```
Flash::warning(“Advertencia: No ha iniciado sesión en el sistema”);
```

Flash::show($name, $text)

...

Session

La clase Session es para facilitar el manejo de la sesiones.

Session::set($index, $value, $namespace='default')

Crear o especifica el valor para un índice de la sesión actual.

```
<?php Session::set('usuario', 'Administrador'); ?>
```

Session::get($index, $namespace='default')

Obtener el valor para un índice de la sesión actual.

```
<?php
Session::get('usuario');//retorna 'Administrador'
?>
```

Session::delete($index, $namespace='default')

Elimina el valor para un índice de la sesión actual.

```
<?php Session::delete('usuario'); ?>
```

Session::has($index, $namespace='default')

Verifica que esté definido el índice en la sesión actual.

```
<?php
Session::has('id_usuario');//retorna false.
?>
```

NOTA: $namespace es un espacio individual en el cual se pueden contener las variables de sesión, permitiendo evitar colisiones con nombres de variables.

Load

La clase load permite la carga de librerías en KumbiaPHP.

Load::coreLib($lib)

Permite cargar una librería del núcleo de KumbiaPHP.

```
<?php
// Carga la librería cache
Load::coreLib('cache');
?>
```

Load::lib($lib)

Permite cargar una librería de aplicación. Las librerías de aplicación se ubican en el directorio "app/libs".

```
<?php
// Carga el archivo app/libs/split.php
Load::lib('split');
?>
```

En caso de que no exista la librería intenta cargar una del núcleo con el nombre indicado.

```
<?php
/* Carga el archivo "app/libs/auth2.php" si existe, en caso
contrario, cargará la libreria del núcleo auth2 */
Load::lib('auth2');
?>
```

Para agrupar librerias debes colocarlas en un subdirectorio y anteceder el nombre del directorio en la ruta al momento de cargarla.

```
<?php
// Carga el archivo app/libs/controllers/auth_controller.php
Load::lib('controllers/auth_controller.php');
?>
```

Load::model($model)

Carga e instancia el modelo indicado. Retorna la instancia del modelo.

```
<?php
// Carga e instancia el modelo usuario.php
$usuario = Load::model('usuario');
?>
```

Para agrupar modelos debes colocarlos en un subdirectorio y anteceder el nombre del directorio en la ruta al momento de cargarlo.

```
<?php
// Carga e instancia el modelo 'partes_vehiculo/motor.php"
$motor = Load::model('partes_vehiculo/motor.php');
?>
```

Auth2

Esta clase permite manejar autenticación de usuarios, con este fin se utilizan adaptadores para tipos especializados de autenticación.

Solicitando un adaptador

Para solicitar un adaptador se hace uso del método estático "factory", dicho método recibe como argumento el tipo de adaptador a utilizar. En caso de no indicarse el tipo de adaptador se utiliza el adaptador predeterminado.

Ejemplo:

```
<?php
$auth = Auth2::factory('model');
?>
```

Los siguientes adaptadores se encuentran implementados:

- **Model**: permite tomar como fuente de datos un modelo ActiveRecord. Debe indicarse en el argumento de factory "model".

Adaptador predeterminado

El adaptador predeterminado es "model", sin embargo esto se puede modificar utilizando el método estático *setDefault*.

```
$adapter (string): nombre de adaptador

setDefault($adapter)

Ejemplo:

Auth2::setDefault('model');
```

Cómo trabaja la autenticación

El método *identify* verifica si existe una sesión autenticada previa, en caso contrario toma de *$_POST* el usuario y clave de acceso, y verifica el usuario y la clave encriptada contra la fuente de datos. De manera predeterminada la clave es encriptada utilizando *md5.*

Para poder efectuar la autenticación debe existir una variable *$_POST['mode']* cuyo valor debe ser "auth".

El formulario para autenticación debe tener la siguiente estructura básica:

```
<?php echo Form::open() ?>
    <input name="mode" type="hidden" value="auth">

    <label for="login">Usuario:</label>
    <?php echo Form::text('login') ?>

    <label for="password">Clave:</label>
    <?php echo Form::pass('password') ?>
<?php echo Form::close() ?>
```

De manera predeterminada Auth2 toma para el nombre de usuario el campo "login" y para la clave el campo "password".

Para poder iniciar una sesión de usuario y realizar la autenticación se debe invocar el método *identify*, sin embargo dependiendo del tipo de adaptador, es necesario especificar ciertos parámetros de configuración.

Adaptador Model

Este adaptador permite utilizar autenticación en base a un modelo que herede de la clase *ActiveRecord*, verificando los datos de autenticación en la base de datos.

setModel()

Establece el modelo ActiveRecord que se utilizará como fuente de datos. De manera predeterminada el modelo que se utilizará como fuente de datos es 'users'.

```
$model (string): nombre de modelo en smallcase

setModel($model)

Ejemplo:
$auth->setModel('usuario');
```

identify()

Realiza la autenticación. Si ya existe una sesión de usuario activa o los datos de usuario son

correctos, entonces la identificación es satisfactoria.

```
return boolean

identify()

Ejemplo:
$valid = $auth->identify();
```

logout()

Termina la sesión de usuario.

```
logout()

Ejemplo:
$auth->logout();
```

setFields()

Establece los campos del modelo que se cargarán en sesión mediante el método *Session::set*. De manera predeterminada se carga el campo “id”.

```
$fields (array): arreglo de campos

setFields($fields)

Ejemplo:
$auth->setFields(array(‘id’, ‘usuario’));
```

setSessionNamespace()

Establece un namespace para los campos que se cargan en sesión.

```
$namespace (string): namespace de sesion

setSessionNamespace($namespace)

Ejemplo:
$auth->setSessionNamespace(‘auth’);
```

isValid()

Verifica si existe una sesión de usuario autenticado.

```
return boolean

isValid()

Ejemplo:
$valid = $auth->isValid();
```

getError()

Obtiene el mensaje de error.

```
return string

getError()

Ejemplo:
if(!$auth->identify()) Flash::error($auth->getError());
```

setAlgos()

Establece el método de encriptación de la clave de usuario.

```
$algos (string): método de encriptación, el nombre coincide con la
función hash de php.

setAlgos($algos)

Ejemplo:
$auth->setAlgos('md5');
```

setKey()

Establece la clave para identificar si existe una sesión autenticada, dicha clave toma un valor booleano "true" cuando la sesión autenticada es válida, asignada mediante el método *Session::set*.

```
$key (string): clave de sesión

setKey($key)

Ejemplo:
$auth->setKey('usuario_logged');
```

setCheckSession()

Indica que no se inicie sesión desde browser distinto con la misma IP.

```
$check (boolean): indicador

setCheckSession($check)

Ejemplo:
$auth->setCheckSession(true);
```

setPass()

Asigna el nombre de campo para el campo de clave. Este campo debe corresponder con el campo de la base de datos y del formulario. De manera predeterminada es “password”.

```
$field (string): nombre de campo que recibe por POST.

setPass($field)

Ejemplo:
$auth->setPass(‘clave’);
```

setLogin()

Asigna el nombre de campo para el campo de nombre de usuario. Este campo debe corresponder con el campo de la base de datos y del formulario. De manera predeterminada es “login”.

```
$field (string): nombre de campo que recibe por POST.

setLogin($field)

Ejemplo:
$auth->setLogin(‘usuario’);
```

Obtener los campos cargados en sesión

Los campos se obtienen por medio del método *Session::get*.

```
$id = Session::get(‘id’);
```

Si se ha especificado un *namespace* de sesión, entonces debe indicarlo al invocar el método.

```
$id = Session::get('id', 'mi_namespace');
```

Ejemplo

La vista:
app/views/acceso/login.phtml

```
<?php echo Form::open() ?>
    <input name="mode" type="hidden" value="auth">

    <label for="login">Usuario:</label>
    <?php echo Form::text('login') ?>

    <label for="password">Clave:</label>
    <?php echo Form::pass('password') ?>
<?php echo Form::close() ?>
```

El controlador:
app/controllers/auth_controller.php

```
<?php
class AuthController extends AppController
{
    public function login()
    {
        // Si se loguea se redirecciona al módulo de cliente
        if(Load::model('usuario')->login()) {
            Router::toAction('usuario/panel');
        }
    }

    public function logout()
    {
        // Termina la sesion
        Load::model('usuario')->logout();
        Router::toAction('login');
    }
}
?>
```

Para validar que el usuario esté autenticado, basta con adicionar en cualquier acción del controlador o en el método *before_filter* el siguiente código:

```
if(!Load::model('usuario')->logged()) {
    Router::toAction('auth/login');
```

```
    return false;
}
```

El modelo:
app/models/usuario.php

```
<?php
// Carga de la libreria auth2
Load::lib('auth2');

class Usuario extends ActiveRecord
{
    /**
     * Iniciar sesion
     *
     */
    public function login()
    {
        // Obtiene el adaptador
        $auth = Auth2::factory('model');

        // Modelo que utilizará para consultar
        $auth->setModel('usuario');

        if($auth->identify()) return true;

        Flash::error($auth->getError());
        return false;
    }

    /**
     * Terminar sesion
     *
     */
    public function logout()
    {
        Auth2::factory('model')->logout();
    }

    /**
     * Verifica si el usuario esta autenticado
     *
     * @return boolean
     */
    public function logged()
    {
        return Auth2::factory('model')->isValid();
    }
```

```
}
?>
```

9 Usar clases externas

10 La Consola

Introducción

La consola, es una herramienta de línea de comandos de KumbiaPHP que permite realizar tareas automatizadas en el ámbito de tu aplicación. En este sentido KumbiaPHP incluye las siguientes consolas: Cache, Model y Controller.

Cada consola está compuesta por un conjunto de comandos, cada comando puede recibir **argumentos secuenciales** y **argumentos con nombre**. Para indicar un argumento con nombre se debe anteceder el prefijo “**--**” al argumento.

Cómo utilizar la Consola

Para utilizar la consola debes ejecutar el despachador de comandos de consola de KumbiaPHP en un terminal, ubicarte en el directorio “**app”** de tu aplicación y ejecutar la instrucción acorde al siguiente formato:

```
php ../../core/console/kumbia.php [ consola] [comando] [arg]
[--arg_nom]=valor
```

Si no se especifica el comando ha ejecutar, entonces se ejecutará el comando “**main**” de la consola.

También es posible indicar la ruta al directorio **app** de la aplicación explícitamente por medio del argumento con nombre “**path**“.

Ejemplos:

```
php ../../core/console/kumbia.php cache clean --driver=sqlite
php kumbia.php cache clean --driver=sqlite --path="/var/www/app"
```

Consolas de KumbiaPHP

Cache

Esta consola permite realizar tareas de control sobre la cache de aplicación.

clean [group] [--driver]

Permite limpiar la cache.

Argumentos secuenciales:

- **group:** nombre de grupo de elementos de caché que se eliminará, si no se especifica valor, entonces se limpiará toda la caché.

Argumentos con nombre:

- **driver:** manejador de caché correspondiente a la caché a limpiar (nixfile, file, sqlite, APC), si no se especifica, entonces se toma el manejador de cache predeterminado.

Ejemplo:

```
php ../../core/console/kumbia.php cache clean
```

remove [id] [group]

Elimina un elemento de la caché.

Argumentos secuenciales:

- **id:** id de elemento en caché.
- **group:** nombre de grupo al que pertenece el elemento, si no se especifica valor, entonces se utilizará el grupo 'default'.

Argumentos con nombre:

- **driver:** manejador de caché correspondiente a la caché a limpiar (nixfile, file, sqlite, APC).

Ejemplo:

```
php ../../core/console/kumbia.php cache remove vista1 mis_vistas
```

Model

Permite manipular modelos de la aplicación.

create [model]

Crea un modelo utilizando como base la plantilla ubicada en "core/console/generators/model.php".

Argumentos secuenciales:

- **model:** nombre de modelo en smallcase.

Ejemplo:

```
php ../../core/console/kumbia.php model create venta_vehiculo
```

delete [model]

Elimina un modelo.
Argumentos secuenciales:

- **model:** nombre de modelo en smallcase.

Ejemplo:

```
php ../../core/console/kumbia.php model delete venta_vehiculo
```

Controller

Permite manipular controladores de la aplicación.

create [controller]

Crea un controlador utilizando como base la plantilla ubicada en ‘core/console/generators/controller.php’.

Argumentos secuenciales:

- **controller:** nombre de controlador en smallcase.

Ejemplo:

```
php ../../core/console/kumbia.php controller create venta_vehiculo
```

delete [controller]

Elimina un controlador.

Argumentos secuenciales:

- **controller:** nombre de controlador en smallcase.

Ejemplo:

```
php ../../core/console/kumbia.php controller delete venta_vehiculo
```

Desarrollando tus Consolas

Para desarrollar tus consolas debes de considerar lo siguiente:

- Las consolas que desarrolles para tu aplicación deben estar ubicadas en el directorio "app/extensions/console".
- El archivo debe tener el sufijo "_console" y de igual manera la clase el sufijo "Console".
- Cada comando de la consola equivale a un método de la clase.
- Los argumentos con nombre que son enviados al invocar un comando se reciben en el primer argumento del método correspondiente al comando.
- Los argumentos secuenciales que son enviados al invocar un comando se reciben como argumentos del método invocado posteriores al primer argumento.
- Si no se especifica el comando a ejecutar, se ejecutará de manera predeterminada el método "main" de la clase.
- Las clases *Load*, *Config* y *Util;* son cargadas automáticamente para la consola.
- Las constantes *APP_PATH, CORE_PATH* y *PRODUCTION;* se encuentran definidas para el entorno de la consola.

Ejemplo:

Consideremos una parte del código de la consola cache cuya funcionalidad fue explicada en la sección anterior.

```
<?php
Load::lib('cache');

class CacheConsole
{
    public function clean($params, $group = FALSE)
    {
        // obtiene el driver de cache
        if (isset($params['driver'])) {
            $cache = Cache::driver($params['driver']);
        } else {
            $cache = Cache::driver()
        }

        // limpia la cache
        if ($cache->clean($group)) {
            if ($group) {
                echo "-> Se ha limpiado el grupo $group", PHP_EOL;
           } else {
                echo "-> Se ha limpiado la cache", PHP_EOL;
           }
        } else {
            throw new KumbiaException('No se ha logrado eliminar el
contenido');
        }
    }
```

```
}
?>
```

Console::input

Este método de la clase *Console* permite leer una entrada desde el terminal, se caracteriza por intentar leer la entrada hasta que esta sea válida.

```
Console::input($message, $values = null)

$message (string): mensaje a mostrar al momento de solicitar la
entrada.
$values (array): conjunto de valores válidos para la entrada.


Ejemplo:

$valor = Console::input('¿Desea continuar?', array('s', 'n'));
```

Apéndices

Integración de jQuery y KumbiaPHP

KumbiaPHP provee de una integración con el Framework de DOM en JavaScript, jQuery

KDebug

KDebug es un nuevo objeto incorporado al plugins de integración KumbiaPHP/jQuery que permite una depuración del código en tiempo de desarrollo. Con solo un parámetro se puede aplicar un log que permite ver en consola (mientras esta esté disponible, sino usará alert) que permite un mejor control de la ejecución.
No es necesario pero si recomendable usar Firebug si se trabaja en Mozilla Firefox o algun navegador que use la consola de WebKit como Google Chrome.

CRUD

Introducción

Este ejemplo nos permitirá de manera sencilla conocer y entender la implementación de un CRUD (Create, Read, Update y Delete en inglés) sin la necesidad de un Scaffold (StandardForm) y un manejo correcto del MVC en KumbiaPHP.

El CRUD de la beta1 sigue funcionando igual en la beta2, pero queda desaconsejado. En la versión 1.0 se podrá usar de las 2 maneras. Y la 1.2 que saldrá junto a la 1.0 sólo se usará lo nuevo y se eliminará lo desaconsejado.

Configurando database.ini

Configurar el archivo databases.ini, con los datos y motor de Base de Datos a utilizar.

Modelo

Crear el Modelo el cual esta viene dado por la definición de una tabla en la BD, para efecto del ejemplo creamos la siguiente tabla.

```
CREATE TABLE menus
(
id            int              unique not null auto_increment,
nombre        varchar(100),
titulo        varchar(100)     not null,
primary key(id)
)
```

Vamos ahora a definir el modelo el cual nos permite interactuar con la BD.

[app]/models/menus.php:

```
<?php
class Menus extends ActiveRecord
{
    /**
      * Retorna los menu para ser paginados
      *
      */
    public function getMenus($page, $ppage=20)
    {
        return $this->paginate("page: $page", "per_page: $ppage",
'order: id desc');
    }
}
```

Controller

El controlador es encargado de atender las peticiones del cliente (ej. browser) y a su vez de darle una respuesta. En este controller vamos a definir todas las operaciones CRUD que necesitamos.

[app]/controllers/menus_controller.php:

```
<?php
/**
* Carga del modelo Menus...
*/
Load::models('menus');

class MenusController extends AppController {
    /**
     * Obtiene una lista para paginar los menus
     */
    public function index($page=1)
    {
        $menu = new Menus();
        $this->listMenus = $menu->getMenus($page);
    }

    /**
     * Crea un Registro
     */
```

```
    public function create ()
    {
        /**
         * Se verifica si el usuario envio el form (submit) y si ademas
         * dentro del array POST existe uno llamado "menus"
         * el cual aplica la autocarga de objeto para guardar los
         * datos enviado por POST utilizando autocarga de objeto
         */
        if(Input::hasPost('menus')){
            /**
             * se le pasa al modelo por constructor los datos del form
y ActiveRecord recoge esos datos
             * y los asocia al campo correspondiente siempre y cuando
se utilice la convención
             * model.campo
             */
            $menu = new Menus(Input::post('menus'));
            //En caso que falle la operación de guardar
            if(!$menu->save()){
                Flash:: error('Falló Operación');
            }else{
                Flash:: valid('Operación exitosa');
                //Eliminamos el POST, si no queremos que se vean en el
form
                Input:: delete();
            }
        }
    }

    /**
     * Edita un Registro
     *
     * @param int $id (requerido)
     */
    public function edit($id)
    {
        $menu = new Menus();

        //se verifica si se ha enviado el formulario (submit)
        if(Input::hasPost('menus')){

            if(!$menu->update(Input::post('menus'))){
                Flash:: error('Falló Operación');
            } else {
                Flash:: valid('Operación exitosa');
```

```
                //enrutando por defecto al index del controller
                return Router::redirect();
            }
        } else {
            //Aplicando la autocarga de objeto, para comenzar la
edición
            $this->menus = $menu->find((int)$id);
        }
    }

    /**
     * Eliminar un menu
     *
     * @param int $id (requerido)
     */
    public function del($id)
    {
        $menu = new Menus();
        if (!$menu->delete((int)$id)) {
                Flash::error('Falló Operación');
        }else{
                Flash::valid('Operación exitosa');
        }

        //enrutando por defecto al index del controller
        return Router::redirect();
    }
}
```

Vistas

Agregamos las vistas...

[app]/views/menus/index.phtml

```
<div class="content">
    <?php echo View::content(); ?>
    <h3>Menus</h3>
    <ul>
    <?php foreach ($listMenus->items as $item) : ?>
    <li>
        <?php echo Html::linkAction("edit/$item->id/", 'Editar') ?>
        <?php echo Html::linkAction("del/$item->id/", 'Borrar') ?>
        <strong><?php echo $item->nombre ?> - <?php echo
$item->titulo ?></strong>
```

```
   </li>
   <?php endforeach; ?>
   </ul>

    // ejemplo manual de paginador, hay partial listos en formato
digg, clasic,....
   <?php if($listMenus->prev) echo
Html::linkAction("index/$listMenus->prev/", '<< Anterior |'); ?>
   <?php if($listMenus->next) echo
Html::linkAction("index/$listMenus->next/", 'Próximo >>'); ?>
</div>
```

[app]/views/menus/create.phtml

```
<?php View::content(); ?>
<h3>Crear menu<h3>

<?php echo Form::open(); // por defecto llama a la misma url ?>

       <label>Nombre
       <?php echo Form::text('menus.nombre') ?></label>

       <label>Título
       <?php echo Form::text('menus.titulo') ?></label>

       <?php echo Form::submit('Agregar') ?>

<?php echo Form::close() ?>
```

[app]/views/menus/edit.phtml

```
<?php View::content(); ?>
<h3>Editar menu<h3>

<?php echo Form::open(); // por defecto llama a la misma url ?>

       <label>Nombre
       <?php echo Form::text('menus.nombre') ?></label>

       <label>Título
       <?php echo Form::text('menus.titulo') ?></label>

       <?php echo Form::hidden('menus.id') ?>
       <?php echo Form::submit('Actualizar') ?>

<?php echo Form::close() ?>
```

Probando el CRUD

Ahora solo resta probar todo el código que hemos generado, en este punto es importante conocer el comportamiento de las URL's en KumbiaPHP.

- index es la acción para listar http://localhost/menus/index/

NOTA: index/ se puede pasar de forma implícita o no. KumbiaPHP en caso que no se le pase una acción, buscará por defecto un index, es decir si colocamos: http://localhost/menus/

- create crea un menu en la Base de Datos http://localhost/menus/create/
- Las acciones del y edit a ambas se debe entrar desde el index, ya que reciben el parámetros a editar o borrar según el caso.

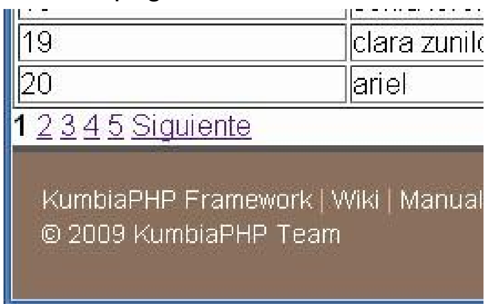
Aplicación en producción

Partials de paginación

Como complemento para el paginador de ActiveRecord, a través de vistas parciales se implementan los tipos de paginación más comunes. Estos se ubican en el directorio "core/views/partials/paginators" listos para ser usados. Son completamente configurables via CSS. Por supuesto, podéis crear vuestros propios partials para paginar en las vistas.

Classic

Vista de paginación clásica.



Resultado Final

Parámetros de configuración:
page: objeto obtenido al invocar al paginador.
show: número de páginas que se mostrarán en el paginador, por defecto 10.
url: url para la acción que efectúa la paginación, por defecto "module/controller/page/" y se envía por parámetro el número de página.

```
View::partial('paginators/classic', false, array('page' => $page,
'show' => 8, 'url' => 'usuario/lista'));
```

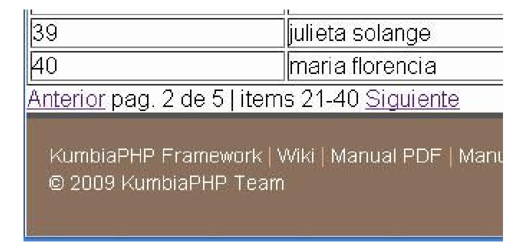
Digg

Vista de paginación estilo digg.

Parámetros de configuración:
page: objeto obtenido al invocar al paginador.
show: número de páginas que se mostrarán en el paginador, por defecto 10.
url: url para la acción que efectúa la paginación, por defecto "module/controller/page/" y se envía por parámetro el número de página.

```
View::partial('paginators/digg', false, array('page' => $page,
'show' => 8, 'url' => 'usuario/lista'));
```

Extended



Resultado Final

Vista de paginación extendida.

Parámetros de configuración:
page: objeto obtenido al invocar al paginador.
url: url para la acción que efectúa la paginación, por defecto "module/controller/page/" y se envía por parámetro el número de página.

```
View::partial('paginators/extended' , false, array('page' => $page,
'url' => 'usuario/lista'));
```

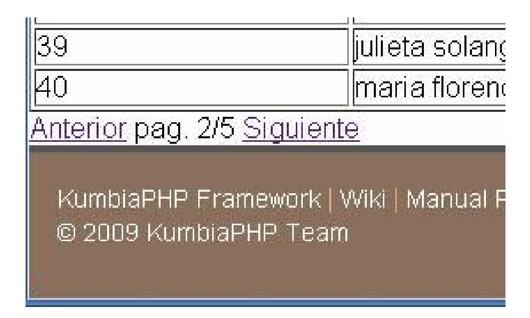
Punbb

Vista de paginación estilo punbb.

Parámetros de configuración:
page: objeto obtenido al invocar al paginador.
show: número de páginas que se mostrarán en el paginador, por defecto 10.
url: url para la acción que efectúa la paginación, por defecto "module/controller/page/" y se envía por parámetro el número de página.

```
View::partial('paginators/punbb' , false, array('page' => $page,
'show' => 8, 'url' => 'usuario/lista'));
```

Simple



Resultado Final

Vista de paginación simple.

Parámetros de configuración:
page: objeto obtenido al invocar al paginador.
url: url para la acción que efectúa la paginación, por defecto "module/controller/page/" y se envía por parámetro el número de página.

```
View::partial('paginators/simple' , false, array('page' => $page,
'url' => 'usuario/lista'));
```

Ejemplo de uso

Supongamos que queremos paginar una lista de usuarios.

Para el modelo Usuario en models/usuario.php:

```
<?php
class Usuario extends ActiveRecord
{
    /**
     * Muestra los usuarios de cinco en cinco utilizando paginador
     *
     * @param int $page
     * @return object
     **/
    public function ver($page=1)
    {
        return $this->paginate("page: $page", 'per_page: 5');
    }
}
?>
```

Para el controlador UsuarioController en controllers/usuario_controller.php:

```
<?php
Load::models('usuario');

class UsuarioController extends AppController
{

    /**
     * Accion de paginacion
     *
```

```
     * @param int $page
     **/
    public function page($page=1)
    {
        $Usuario = new Usuario();
        $this->page = $Usuario->ver($page);
    }
}
?>
```

Y en la vista views/usuario/page.phtml

```
<table>
<tr>
 <th>Id</th>
 <th>Nombre</th>
</tr>
<?php foreach($page->items as $p): ?>
<tr>
 <td><?php echo $p->id; ?></td>
 <td><?php echo $p->nombre; ?></td>
</tr>
<?php endforeach; ?>
</table>

<?php View::partial('paginators/classic', false, array('page' =>
$page)); ?>
```

Auth

Beta1 a Beta2

Deprecated

Métodos y clases que se usaban en versiones anteriores y que aun funcionan. Pero que quedan desaconsejadas y que no funcionaran en el futuro (próxima beta o versión final):

Posiblemente habrá 2 versiones:
beta2 con lo deprecated para facilitar migración
beta2.2 sin lo deprecated más limpia y rápida, para empezar nuevas apps

Flash::success() ahora Flash::valid()
Flash::notice() ahora Flash::info()

ApplicationController ahora AppController (con sus respectivos cambios de métodos)
….

Usar $this->response = 'view' o View::response('view') para no mostrar el template.
Ahora View::template(NULL) el View::response() sólo se usa para elegir formatos de vista alternativos.

Lista de cambios entre versiones:$^{\text{si no se especifica beta1 es que es compatible en ambos casos}}$

Application
ControllerBase$^{0.5}$ => ApplicationController$^{\text{beta1}}$ => AppController$^{\text{beta2}}$
public function init$^{0.5}$ => protected function initialize$^{\text{beta2}}$
render_view$^{0.5}$ => View::select$^{\text{beta2}}$

Models
public $mode$^{0.5}$ => public $database$^{\text{beta1 y beta2}}$

Callbacks
public function initialize$^{0.5}$ => protected function initialize$^{\text{beta2}}$
public function finalize$^{0.5}$ => protected function finalize$^{\text{beta2}}$
public function before_filter$^{0.5}$ => protected function before_filter$^{\text{beta2}}$
public function after_filter$^{0.5}$ => protected function after_filter$^{\text{beta2}}$

boot.ini$^{\text{se elimina en beta2}}$
kumbia / mail / libchart$^{0.5}$ => se elimina los prefijos$^{\text{beta1}}$
extensions$^{0.5}$ => libs$^{\text{beta1}}$

Input::
$this->has_post$^{0.5}$ => Input::hasPost$^{\text{beta2}}$
$this->has_get$^{0.5}$ => Input::hasGet$^{\text{beta2}}$
$this->has_request$^{0.5}$ => Input::hasRequest$^{\text{beta2}}$
$this->post$^{0.5}$ => 'Input::post$^{\text{beta2}}$

$this->get[0.5] => 'Input::get[beta2]
$this->request[0.5] => 'Input::request[beta2]

View::
$this->cache[0.5] => View::cache[beta2]
$this->render[0.5] => 'View::select[beta2]
$this->set_response[0.5] => View::response[beta2]
content()[0.5] => View::content()[beta2]
render_partial[0.5] => View::partial[beta2]

Router::
$this->route_to[0.5] => 'Router::route_to[beta1 y beta2]
$this->redirect[0.5] => Router::redirect[beta2]

Html::
img_tag[0.5] => 'Html::img[beta2]
link_to[0.5] => 'Html::link[beta2]
link_to_action[0.5] => 'Html::linkAction[beta2]
stylesheet_link_tags[0.5] => 'Html::includeCss[beta2]

Ajax::
form_remote_tag[0.5] => 'Ajax::form[beta2]
link_to_remote[0.5] => 'Ajax::link[beta2]

Form::
end_form_tag[0.5] => 'Form::close[beta2]
form_tag[0.5] => 'Form::open[beta2]
input_field_tag[0.5'] => 'Form::input[beta2]
text_field_tag[0.5] => 'Form::text[beta2]
password_field_tag[0.5] => 'Form::pass[beta2]
textarea_tag[0.5] => 'Form::textarea[beta2]
hidden_field_tag[0.5] => 'Form::hidden[beta2]
select_tag[0.5] => 'Form::select[beta2]
file_field_tag[0.5] => 'Form::file[beta2]
button_tag[0.5] => 'Form::button[beta2]
submit_image_tag[0.5] => 'Form::submitImage[beta2]
submit_tag[0.5] => 'Form::submit[beta2]
checkbox_field_tag[0.5] => 'Form::check[beta2]
radio_field_tag[0.5] => 'Form::radio[beta2]

Tag::
javascript_include_tag[0.5] => 'Tag::js[beta2]
stylesheet_link_tag[0.5] => 'Tag::css[beta2]

Cambio en las rutas entre versiones:

0.5 => 1.0 beta1

'/apps/default' => '/app',
'/apps' => '',
'/app/controllers/application.php' => '/app/application.php',
'/app/views/layouts' => '/app/views/templates',
'/app/views/index.phtml' => '/app/views/templates/default.phtml',
'/app/views/not_found.phtml' => '/app/views/errors/404.phtml',
'/app/views/bienvenida.phtml' => '/app/views/pages/index.phtml',
'/app/helpers' => '/app/extensions/helpers',
'/app/models/base/model_base.php' => '/app/model_base.php',
'/app/models/base/' => '',
'/cache' => '/app/cache',
'/config' => '/app/config',
'/docs' => '/app/docs',
'/logs' => '/app/logs',
'/scripts' => '/app/scripts',
'/test' => '/app/test',

1.0 beta1 => 1.0 beta2

…

Cambiados:

Session::isset_data() ahora Session::has()
Session::unset_data() ahora Session::delete()

Glosario

CRUD = Create Read Update Delete (Crear Leer Actualizar Borrar)
ABM
MVC = Model View Controller (Modelo Vista Controlador)
HTML = HyperText Markup Language (Lenguaje de Marcado de HiperTexto)
SQL = Structured Query Language (Lenguaje de Consulta Estructurado)
….